European Sixth Framework Network of Excellence FP6-2004-IST-026854-NoE

Deliverable D7.3

Large scale Management Interim Report

For more information on this document or the EMANICS Project, please contact:

Dr. Olivier Festor
Technopole de Nancy-Brabois - Campus scientifique
615, rue de Jardin Botanique - B.P. 101
F-54600 Villers Les Nancy Cedex
France
Phone: +33 383 59 30 66
Fax: +33 383 41 30 79
E-mail: <olivier.festor@loria.fr>

Document Control

Title:	Large scale Management Interim Report
Type:	Public
Editor(s):	Radu State
E-mail:	radu.state@loria.fr
Author(s):	WP7 Partners
Doc ID:	D7.3

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
0.1	2007-07-03	H. Tran, J. Schönwälder	Initial version of a LaTeX template
0.2	2008-03-15	R. State	Integration of action leader contributions
0.3	2008-04-07	O. Festor	Integration of Quality check comments

Contents

1 Executive Summary

Management performance evaluation means assessment of scalability, complexity, accuracy, throughput, delays and resource consumptions.This deliverable reports the main scientific results obtained in work-package 7 of the Emanics NoE. These activities started in July 2007 as a result of an open call for proposals and addressed three main topics: the analysis of network management trace collections, the distributed and scalable security monitoring and finally a new configuration paradigm based on voluntary cooperation. The reported content has been also the subject of submissions to several outstanding scientific conferences.

2 Introduction

The activities in the work-package 7 have been driven by three main tasks, that emerged from an open call for proposals. In July 2007, the three selected proposals were addressing relevant topics to the scalability of network management. The first topic, addressed in the activity NMTA(Network Management Trace Analysis), is centered on precisely defining the key concepts and terminology for SNMP trace analysis. This activity has resulted in many drafts, submitted to the IETF by the main task leader.The third chapter in this deliverable reports on this activity. The second topic is addressing the scalable and distributed security monitoring.In the SNID task (Scalability of Network Intrusion Detection), we addressed the issues on how flow based monitoring is dependent of the network scale and sampling approach and covered the algorithmic graph based methods required to monitor large network telescopes and honeypots. We report this work in the fourth chapter of this deliverable. The third topic is concerned with a new configuration paradigm based on voluntary peer to peer contribution.This work proposes a radically new theoretical concept of using the notion of promise in developing large scale configuration operations.We cover this activity in the fourth chapter in this deliverable. For an overall view on the activities and the involved participants, we summarize them in the following:

- NMTA: Network Management Trace Analysis (9 months, partners: JUB, UT, PSNC)
- SNID: Scalability of Network Intrusion Detection (9 months, partners: UT, UZH, PSNC, JUB (limited resources, observer), UniBW, INRIA)
- SNPS: Scaling of Network Peer Services employing Voluntary Cooperation (18 months, Partners: HIO, KTH, INRIA)

3 Network Management Trace Analysis (NMTA)

The Simple Network Management Protocol (SMMP) was introduced in the late 1980s. Since then, several protocol changes have taken place, which have eventually led to what is known today as the SNMP version 3 framework (SNMPv3) [1, 2]. Extensive use of SNMP has led to significant practical experience by both network operators and researchers. However, up until now only little research has been done on characterizing and modeling SNMP traffic.

Since recently, network researchers are in the possession of network traces, including SNMP traces captured on operational networks. The availability of SNMP traces enables research on characterizing and modeling real world SNMP traffic. However, experience with SNMP traces has shown that the traces must be large enough in order to make proper observations. A more detailed motivation for collecting SNMP traces and guidelines how to capture SNMP traces can be found in [3].

Unfortunately, the analysis of large SNMP traces can take a large amount of processing time. Therefore, it is often desirable to focus the analysis on smaller, relevant sections of a trace. This in turn requires a proper way to identify these smaller sections of a trace. This document describes a number of identifiable sections within a trace which make specific research on these smaller sections more practical. The following figure shows the various sections of traces and how they relate to each other.

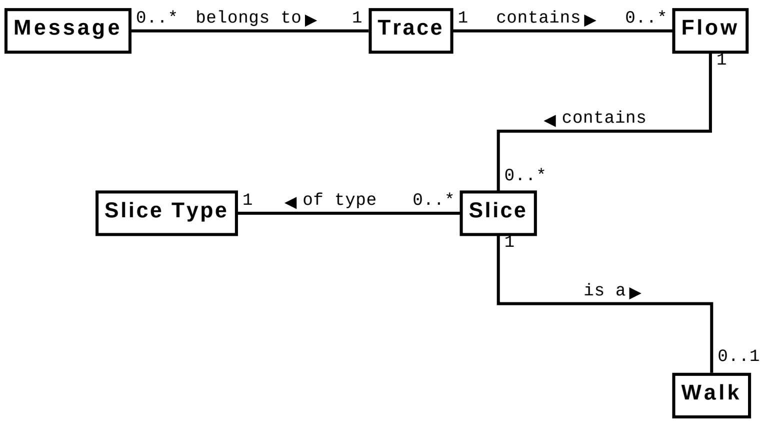


Figure 1: SNMP, Trace Analysis, SNMP Message Separation

This document defines the various entities (boxes) shown in the above figure. These definitions can be implemented by tools that can split SNMP traces into smaller sections for further analysis.

The most central entity in the figure above is an SNMP trace, consisting of a potentially large set of SNMP messages. An SNMP trace is the result of recording SNMP traffic on a specific network for a specific time duration. Such a trace may, depending on the number of hosts in the respective network, contain SNMP messages exchanged between possibly many different SNMP engines. The messages contained in a trace may be represented in different formats. For the purpose of this document, the simple comma separated values (CSV) format defined in [3] contains sufficient information to split a trace into smaller sections.

The SNMP messages belonging to an SNMP trace may have been exchanged between many different SNMP engines running on different hosts. Therefore, a first obvious way of separating a trace into smaller sets of SNMP messages is the separation of a trace into flows. Each flow contains only those SNMP messages of an SNMP trace that have been exchanged between two network endpoints. Such a separation may be necessary in case one wants to analyze specific SNMP traffic characteristics (e.g., number of agents managed by a management station) and wants to rule out network endpoint specific behaviour (e.g., different SNMP management stations may have different polling configurations).

Flows within traces can still be quite large in terms of the number of messages they contain. Therefore, it may be necessary to split a flow into even smaller sections called slices. A slice contains all SNMP messages of a given flow that are related to each other in time and referenced information. Splitting a flow into slices makes it possible to separate SNMP messages within traces that belong to each other, like for example all messages that belong to a single polling instance involving a single manager and a single agent.

A slice may contain, for instance, the exchanged SNMP messages between an agent and a manager, which polls that agent in a single polling instance. The manager may be configured to poll that agent every once in a while. If the requested information from the agent remains unchanged, then the respective slices of SNMP traffic occurring between this manager and agent will be highly comparable. In such a case the slices will be of the same slice type. Similar slices will thus be considered of the same slice type and incomparable slices will not be of the same slice type.

Besides the fact that each slice is of specific slice type, slices can also be of a specific form with respect to the messages encompassing a slice. For example, slices containing a sequence of linked GetNext or GetBulk requests are commonly called an SNMP walk. Note that only a subset of all slices will be walks.

3.1 Messages

SNMP messages carry PDUs associated with well defined specific protocol operations [4]. The PDUs can be used to classify SNMP messages. Following are a number of definitions that help to classify SNMP messages based on the PDU contained in them. These definitions will be used later on in this document.

Notation 1 *Let M be an SNMP message. We denote the properties of M as follows:*

M.type	=	*operation type of message M (get, getnext, ...)*
M.class	=	*class of message M (according to RFC 3411)*
M.tsrc	=	*transport layer source endpoint of message M*
M.tdst	=	*transport layer destination endpoint of message M*
M.nsrc	=	*network layer source endpoint of message M*
M.ndst	=	*network layer destination endpoint of message M*
M.reqid	=	*request identifier of message M*
M.time	=	*capture timestamp of message M*
M.oids	=	*OIDs listed in varbind list of message M*
M.values	=	*values listed in varbind list of message M*

Note that the properties of a message can be easily extracted from the exchange formats defined in [3].

Definition 1 *A **read request message** is a message M containing a PDU of type GetRequest, GetNextRequest, or GetBulkRequest.*

Definition 2 *A **write request message** is a message M containing a PDU of type SetRequest.*

Definition 3 *A **notification request message** is a message M containing a PDU of type InformRequest.*

Definition 4 *A **notification message** is a message M containing a PDU of type Trap or InformRequest.*

Definition 5 *A **request message** is a message M which is either a read request message, a write request message, or a notification request message.*

Definition 6 *A **response message** is a message M containing a PDU of type Response or of type Report.*

Note that Report messages are treated like Response messages since the SNMPv3 specifications currently use Report messages only as an error reporting mechanism, always triggered by the processing of some request messages. In case future SNMP versions or extensions use Report messages without having a request triggering the generation of Report messages, we may have to revisit the definition above.

Definition 7 *A **non-response message** is a message M which is either a read request message, a write request message, or a notification message.*

Definition 8 *A **command message** is a message M which is either a read request message or a write request message.*

Definition 9 *A set of **command group messages** consists of all messages M satisfying either of the following two conditions:*

1. *M is a command message*
2. *M is a response message and there exists a command message C such that the following holds:*

$$\begin{array}{lcl} M.reqid & = & C.reqid \\ M.tdst & = & C.tsrc \\ M.tsrc & = & C.tdst \\ (M.time - C.time) & < & t \end{array}$$

The parameter t defines a maximum timeout for response messages.

This definition requires that the response message originates from the transport endpoint over which the request message has been received. This is not strictly required by SNMP transport mappings and in particular the UDP transport mapping allows to send responses from different transport endpoints. While sending response messages from a different transport endpoint is legal, it is also considered bad practice causing interoperability problems since several management systems do not accept such messages.

It was decided to require matching transport endpoints since doing so significantly simplifies the procedures below and avoids accidentally confusing requests and responses. Implementations responding from different transport endpoints will lead to (a) a larger number of requests without related responses (and likely no retries) and (b) a similarly large number of response messages without a matching request. If such behavior can be detected, the traces should be investigated and if needed the transport endpoints corrected.

Definition 10 *A set of* ***notification group messages*** *consists of all messages M satisfying either of the following two conditions:*

1. *M is a notification message*
2. *M is a response message and there exists a notification request message N such that the following holds:*

$$\begin{array}{lcl} M.reqid & = & N.reqid \\ M.tdst & = & N.tsrc \\ M.tsrc & = & N.tdst \\ (M.time - N.time) & < & t \end{array}$$

The parameter t defines a maximum timeout for response messages.

We again require that the transport endpoints match for notification group messages.

3.2 Traces and Flows

Traces are (large) sets of SNMP messages that are the result of recording SNMP traffic using a single traffic recording unit (e.g., using tcpdump) on a network segment carrying traffic of one or more managers and agents. Traces being used in the remainder of this document may be altered as a result of anonymization, which may result in some message information loss.

Definition 11 *An SNMP* ***trace*** *(or short trace) T is an ordered set of zero or more SNMP messages M. All messages M in T are chronologically ordered according to the capture time stamp M.time.*

Definition 12 *A **flow** F is the set of messages of an SNMP trace T with the following properties:*

1. *All response messages originate from a single network endpoint.*
2. *All non-response messages originate from a single network endpoint.*
3. *All messages are either command group messages or notification group messages.*

Subsequently, we call flows containing only command group messages command flows. Similarly, we call flows containing only notification group messages notification flows.

Note that it is possible that response messages of a trace cannot be classified to belong to any flow. This can happen if request messages triggering the response messages were not recorded (for example due to asymmetric routing) or because response messages were originating from transport endpoints different from the endpoint used to receive the associated request message.

Definition 13 *A **flow initiator** is the network endpoint of the two endpoints involved in a flow, which is responsible for sending the first non-response message.*

Notation 2 *Let F be a flow as defined above. We denote the properties of F as follows:*

F.type	=	*type of the flow F (command/notification)*
F.nsrc	=	*network layer source endpoint of F*
F.ndst	=	*network layer destination endpoint of F*
F.start	=	*time stamp of the first message in F*
F.end	=	*time stamp of the last message in F*

This definition of a flow is mostly consistent with the definition of a flow used in [5]. The difference is that the tool used to generate the data reported in [5] did only require that the network layer source endpoint of the response messages matches the destination network layer endpoint of the associated request messages.

3.3 Slices

Flows are made up of smaller sets of messages that are related to each other. Such a subset of messages from a single flow will be considered a slice of a flow.

Definition 14 *A **slice** S is a subset of messages in a flow F for which the following properties hold:*

1. *All messages are exchanged between the same two transport endpoints (a single transport endpoint pair).*
2. *All non-response messages must have a PDU of the same type.*

3. *All messages with a PDU of type Get, Set, Trap, or Inform must contain the same set of OIDs.*
4. *Each GetNext or GetBulk message must either contain the same set of OIDs or they must be linked to the chronologically last response of the previous request, that is the request must contain at least one OID that has been contained in the (repeater) varbind list of the chronologically last response message of a previous request message.*
5. *All Response messages must follow a previous request message that is part of the same slice.*
6. *For any two subsequent request messages Q1 and Q2 with Q1.time $<$ Q2.time, the following condition must hold:*

$$(Q2.time - Q1.time) < e$$

The parameter e defines the maximum time between two non-response messages that belong to a slice. This parameter should be chosen such that unrelated requests within a flow are not considered to be of the same slice. Unrelated requests are those that, for instance, belong to different polling instances. The parameter e should therefore be larger than the retransmission interval in order to keep retransmissions within a slice and smaller than the polling interval used by the slice initiator.

Definition 15 *A **slice initiator** is one of the two transport endpoints involved in a slice, which is responsible for sending the chronologically first non-response class message.*

Notation 3 *A slice S has several properties. We introduce the following notation:*

S.type	=	*type of non-response messages in S*
S.tsrc	=	*transport layer endpoint of initiator of S*
S.tdst	=	*transport layer endpoint of non-initiator of S*
S.start	=	*time stamp of the chronologically first message in S*
S.end	=	*time stamp of the chronologically last message in S*
S.prefix	=	*prefix of S (see below)*

Definition 16 *Two slides A and B of a given flow F are **concurrent** at time t if A.start $\leq$ t $\leq$ A.end and B.start $\leq$ t $\leq$ B.end. The concurrency level F.clevel(t) of a flow F at time t is given by the number of concurrent slices of F at time t. The concurrency level of a manager identified by the network address addr at time t is given by the sum of the flow currency levels F.clevel(t) for all flows originating from addr, that is F.nsrc = addr.*

Definition 17 *Two slides A and B of a given flow F are called **delta time serial** if (B.start - A.end) $<$ delta.*

Table 1: Slice representing a two-column table walk

Message	Direction	PDU type	OIDs
0	A → B	GetNext Request	alpha, beta
1	B → A	Response	alpha.0, beta.0
2	A → B	GetNext Request	alpha.0, beta.0
3	B → A	Response	alpha.1, beta.1
4	A → B	GetNext Request	alpha.1, beta.1
5	B → A	Response	gamma.0, delta.0

3.4 Slice Prefix

As noted in the beginning of this document, it is desired that slices can be tested for equality/comparability. This is where the slice prefix comes in. The slice prefix has as a sole purpose to provide one of the means to compare slices. Using the slice prefix and a few other parameters (which will be discussed later on in this document) of a number of slices, one can determine which slices should be considered 'equal' and which of them are incomparable. This will assist in the process of finding potentially other relations.

The slice prefix is a set of OIDs. This set is constructed based on the messages that make up a single slice. So, for example, a slice that is the result of a manager requesting the contents of a particular table (with OID alpha) on an agent using a simple single varbind GetNext walk, starting at the table OID alpha, shall yield a slice prefix which consists of the OID alpha.

Because the aim is to compare various slices using the slice prefix (along some other characteristics of a slice), this implicitly suggests the need to know whether a number of slices are the result of the same behaviour (i.e., specific configuration) of the initiating party of these slices. For example, one may want to know whether a number slices that involve a single manager and a single agent were the result of just one specific configuration of that manager. Multiple slices, that may all be initiated by that same manager and each slice possibly occurred in different polling instances, may in fact be the result of the same specific configuration of that particular manager. So, since in this case the specific configuration of the manager is only relevant for determining the behaviour, the slice prefix should be constructed based on OIDs in messages originating from that manager only. More generally, only the messages within slices that are sent by the initiating party (the non-response messages) are considered for the determination of the respective slice prefix of a slice.

The resulting set of OID prefixes will represent the behaviour of the respective initiating party of that slice. This allows us to compare different slices.

Following is a short introductory example which depicts what a slice could consist of and how one could determine the slice prefix in such a general case.

Consider the case of a single manager A polling a specific agent B. More specifically, the manager A is configured to retrieve the complete contents of two columns alpha and beta of a some table. The resulting slice may contain the following messages:

The manager starts with a GetNext request referencing two OIDs, alpha and beta. The agent B replies in message 1 with the first items of each of the referenced columns. The manager in turn goes on obtaining data from these two columns until it receives message 5, which indicates that the manager has received all of the data from the two columns.

It can be easily concluded that the manager was configured to retrieve the contents of the two columns alpha and beta (the slice prefix). A different slice involving the same manager and agent and that is again the result of the same configuration of the manager, should be considered 'equal' to this one because the two slices are the result of the same behaviour. It should however be mentioned that such a second slice might contain a different number of messages, since the contents of the tables on the agent side might have changed over time. This underlines the previously made remark that only the messages originating from the initiating party should be considered in this process, because they will (in such a scenario) always illustrate the same behaviour of the initiating party.

The previous example now makes it possible to give a more formal definition of a slice prefix.

Assume the following: Mnon_resp1 and Mnon_resp2 are two consecutive non-response messages of a slice (which have unequal request identifiers) and that Mresp1 and Mresp2 represent any response message to each of the respective non-response messages.

Definition 18 *A **slice prefix** P is a set of OID prefixes derived from the OIDs contained in the non-response messages of a single slice. This set P consists of the following OIDs: Each OID x in Mnon_resp2 of a slice that is not in any response Mresp1 to the previous non-response message Mnon_resp1 (where Mresp1.time $<$ Mnon_resp2.time for each of these response messages in Mnon_resp1) and x is not already in P and there exists no OID in P that makes up a part (starting from the beginning) of x.*

This definition states that all OIDs in the first non-response message are considered part of the resulting slice prefix P. In addition to that, P also contains those OIDs that have been newly introduced in non-response messages (that occurred later than the first one). Newly introduced OIDs are considered as such if they were not included in any of the responses (that occurred before the non-response message in consideration) to the chronologically last preceding non-response message.

Following is an example to illustrate the algorithm just described: Consider the case that a single manager A polling an agent B. More specifically, the manager A is programmed to retrieve the complete contents of two single column tables alpha and beta. Besides that, the manager now also requests the sysUpTime in the first request the manager sends to B. A resulting slice may contain the following messages:

Determining the slice prefix for this slice goes as follows: At the start, the slice prefix P is empty. The algorithm starts looking for the first non-response message, which is message 0. Then, it tests the OIDs contained in message 0 for equality with any OIDs in P. Since no matching OIDs can be found in P and the three referenced OIDs in message 0 are all different, the algorithm adds the three OIDs of message 0 to P.

The algorithm then goes on with the message 1, which is a response. So it proceeds to look at message 2. Since message 2 contains only OIDs that can be found in message 1 (a response to the previous non-response), the algorithm will not consider the OIDs in message 2 any further. The same goes for the remainder of the non-response messages in this slice.

Table 2: Slice prefix computation example #1

Message	Direction	PDU type	OIDs
0	A → B	GetNext Request	sysUpTime, alpha, beta
1	B → A	Response	sysUpTime.0, alpha.0, beta.0
2	A → B	GetNext Request	alpha.0, beta.0
3	B → A	Response	alpha.1, beta.1
4	A → B	GetNext Request	alpha.1, beta.1
5	B → A	Response	gamma.0, delta.0

Table 3: Slice prefix computation example #2

Message	Direction	PDU type	OIDs
0	A → B	GetNext Request	alpha, beta
1	B → A	Response	alpha.0, beta.0
2	A → B	GetNext Request	alpha, beta
3	B → A	Response	alpha.0, beta.1
4	A → B	GetNext Request	beta.1, alpha.0, sysUpTime
5	B → A	Response	gamma.0, alpha.1 sysUpTime.0
6	A → B	GetNext Request	alpha.1
7	B → A	Response	delta.0

This example slice would therefore result in a slice prefix consisting of the OID prefixes sysUpTime, alpha, beta.

Following is a more elaborate slice for which the slice prefix is determined. Consider again the case that a single manager A is set to poll a specific agent B. Manager A is programmed to retrieve some values from B. However, in this case the referenced tables do not have an equal length. Besides that, the manager also requests the sysUpTime in every few requests the manager sends to B. The resulting set of messages within a single slice of this flow may contain the following messages:

Determining the slice prefix for this slice goes as follows: At the start, the slice prefix P is empty. Just as in the previous example, the algorithm analyses the first non-response message (message 0) first. Since P is empty at this point and the two OIDs alpha and beta are different, the OIDs alpha and beta will be added to P.

The second non-response message (message 2) contains two OIDs that cannot be found in the response to the first non-response; they may therefore be added to P. However, P already contains both OIDs, so they will not be added. It should be noted here that this non-response message is probably a retransmission of the first one. Also, it appears that the response to this non-response yields a different result compared to the response to the initial non-response message. This may be caused by a change in the data at the agent side.

Message 4 is the next non-response message. It contains three OIDs, of which two are exactly the same compared to the previous response message (message 3), even though the order is different. A third OID (sysUpTime) in this message cannot be found in message 3, neither can it be found in P. Hence, OID sysUpTime is added to P.

Message 6 is the last non-response message and contains just a single OID that can also be found in message 5, the response to the previous non-response message. Therefore, the single OID in message 6 is not considered any further. It should be noted here that the data retrieval process has apparently reached the end of the table with OID beta, which has resulted in a response containing the lexicographically next data item gamma.0.

After message 6 have all non-response messages been considered in this slice. Even though the order of comparable OIDs within a certain non-response and the previous response may be different (like in non-response message 4 and response message 3), the listed messages still comprise a single slice. The slice also shows the possibility of a manager (A) referencing OIDs that are new compared to a previous response message (like the sysUpTime OID in message 4).

This example slice therefore has a slice prefix P consisting of the OIDs alpha, beta and sysUpTime.

3.5 Slice Equivalence and Slice Type

As described previously, the slice type allows for comparing slices. This means that any number of slices that are of the same slice type may be considered an equivalence class and may therefore be considered to be the result of the same behaviour of the slice initiator.

Definition 19 *Two slices A and B satisfy the binary* ***slice equivalence*** *relation* $A \quad B$ *if the following properties hold:*

1. *All messages in A and B have been exchanged between the same network endpoints.*
2. *All read request messages, write request messages, and notification messages in A and B originate from the same network endpoint.*
3. *All non-response messages in A and B are of the same type.*
4. *The slices A and B have the same prefix, that is A.prefix = B.prefix.*

It can be easily seen that the relation   is reflexive, symmetric, and transitive and thus forms an equivalence relation between slices.

Definition 20 *Let S be a set of slices, then all slices in the equivalence class*

$$[A] = \{s \in S | s \quad A\}$$

with A in S, are of the same ***slice type****.*

3.6 Walks

Definition 21 *A **walk** W is a slice S with the following properties:*

1. *The type of the slice S is either get-next-request or get- bulk-request.*
2. *At least one object identifier in the sequence of requests at the same varbind index must be increasing lexicographically while all object identifiers at the same varbind index have to be non-decreasing.*

Definition 22 *A walk W is a **strict walk** if all object identifiers in the sequence of requests at the same varbind index are strictly increasing lexicographically. Furthermore, the object identifiers at the same index of a response and a subsequent request must be identical.*

Definition 23 *A walk W is a **prefix constrained walk** if all object identifiers at the same index have the same object identifier prefix. This prefix is established by the first request within the walk.*

3.7 Conclusions

The analysis of SNMP traces requires a collection of common and precise definitions in order to establish a basis for producing meaningful and comparable results. This paper provides a collection of basic definitions that have been developed over time and are also being reviewed and discussed within the Network Management Research Group (NMRG) of the IRTF. This paper is a consensed summary of a more detailed document [6] submitted to the NMRG and written to provide an early reference while the work in the NMRG continues and to foster discussion within the broader network management research community.

4 Scalability of Network Intrusion Detection (SNID)

Sophisticated algorithms have been developed in order to detect intrusions into networks. Although they yield impressive results under laboratory conditions, they often do not scale to large real-life networks due to the sheer amount of data transferred in the latter. The topic of the SNID activity is how network-based intrusion detection can be scaled to large-size networks.

A possible approach is to use flow-based techniques. A *flow* summarizes a series of packets sent through the network from a source to a destination. Naturally, information about the contents of the single packets is lost and intrusion detection algorithms based on this information, e.g., most signature-based algorithms, can not be applied. In addition, in some scenarios, even the information reduction offered by the flow-based approach is not sufficient. In fact, ISP and backbone operators have to create the flow data from *sampled* packet streams. Hence, one goal is to design detection algorithms operating on flow data with special regard to the presence of sampling. In particular, we focus on the detection of scans, the detection of botnets and hacked systems that send SPAM, and the exploration of potential hybrid concepts, combining flow-based algorithms with traditional full inspection systems. Since it is generally not yet fully understood how the sampling affects the performance of the detection, a first step toward this goal is to compare traffic characteristics of large networks employing different sampling rates.

Another approach for very large networks is to use *distributed monitoring*. When following this approach, questions about the correlation of the data reported by the different monitors arise, for example: Do all management agents observe the same type of events? Can a temporal behavior of the whole monitoring network be observed?

In the following, we report on performed work. In section 4.1, we investigate how flow-based traffic characteristics behave under the perspective of two main analysis dimensions: network-scale and sampling ratios. We analyze flow-based traces from different scales networks (University of Twente, SURFnet and GÉANT), which use different sampling ratios, in order to look for similarities and differences.

In section 4.2, we propose a new distributed monitoring approach based on the notion of centrality of a graph and its evolution in time. We consider an activity profiling method for a distributed monitoring platform and illustrate its usage in two different target deployments. The first one concerns the monitoring of a distributed honeynet, whilst the second deployment target is the monitoring of a large network telecope.

In section 4.3, we consider the question, whether the scalability advantages of flow-based Intrusion Detection may be combined with the accuracy of traditional packet-based approaches. We describe an evaluation environment, mechanisms to combine the two approaches, and elaborate on test scenarios, in which resource and detection efficiency will be evaluated.

In addition, an annotated bibliography with papers about large-scale, flow-based and sample-based intrusion detection has been created. It is briefly introduced in section 4.4.

4.1 Characterization of IP flows at different network scales and sampling ratios

In LANs, the tools used for intrusion detection, and for network management in general, can often rely on simple packet-based approaches. However such approaches are not anymore possible when the object of study is a major Internet backbone or an IP WAN. The bit rates of several Gbps found in such networks make packet-based analysis very complicated and flow-based technology is used instead. In the following, we rely on Cisco's NetFlow definition of a flow, especially in its version 5 format [7], in which flows are usually defined as source/destination address/port and IP protocol.

The study of traffic characteristics in IP WANs based on flows has risen the attention of the scientific community since long time ago. Common characteristic of previous studies is focusing the attention on a temporal characterization of the traffic, both as time series analysis of metrics of interest as well as analysis of multiple traces collected over time. Moreover, the majority of the authors present results concerning a single network. Only [8], that presents a mainly packet-based analysis in which flows are recreated a posteriori on the basis of extensive TCP/IP header traces, considers traces captured from different locations, but always on academic and research monitoring sites.

We think that these approaches, both historical analysis of traces and comparison of diverse observation points, even if suitable for highly detailed studies, are missing one important dimension of analysis. In the majority, they present isolated studies, in which only one network is considered. On the other hand, in case they are proposing a comparison, only networks of similar usage are taken into account, for example backbones or campus networks [9].

In our studies, we decided to more deeply investigate traffic characteristic on *network basis*, more specifically collecting traces covering the same time interval but belonging to different networks scales. Moreover, the considered networks cover different usage domains with considerable diversity in daily traffic. In this paper we give a comparative analysis concerning traces collected over a 48 hours period in three different locations, namely the University of Twente, the Dutch national research network SURFnet [10], and the European research network GÉANT [11]. In addition, we also consider different packet sampling ratios used in those networks: 1:1 (UT), 1:100 (SURFnet), and 1:1000 (GÉANT).

Our work focuses on the research question: *do the traces collected on so different measurement points and with different sampling ratios show the same characteristics?* We also aim to offer an insight on the networks traffic patterns from an analytic point of view, describing the problems a researcher can face during the measuring and analysis processes and, whenever it is possible, the solutions we adopted.

The section is structured as follows. Section 4.1.1 describes the measurement setup and how the trace collection process has taken place at the University of Twente, focusing the attention on the problems that can be encountered during the collecting and post-processing steps. In Section 4.1.2 we do a comparative analysis of multiple flow traffic characteristics from different traces and show commonalities and differences between them. Finally, Section 4.1.3 presents our conclusions as well as some topics of interest for our future research.

Organizational network	*Sampling ratio*	*Collected data volume (GB)*
UT	1:1	175
SURFnet	1:100	222
GÉANT	1:1000	38

Table 4: Packet sampling ratios and collected data volume per organizational network.

4.1.1 Measurement Setup

This section presents how network traces have been collected from different organizational networks as well as the post-collection steps needed in order to have suitable data for further analysis.

Collecting network traces The traces have been collected over a period of a week (Jul 26, 2007 - Aug 3, 2007) from the University of Twente (UT) network, SURFnet [10], and GÉANT [11]. The networks provide services to organizations with different sizes: enterprise-scale, national-scale and continental-scale, respectively. The UT is a /16 network with an average incoming/outgoing traffic of 500Mbps; SURFnet offers optical path interconnectivity to the major research institutions in the Netherlands; finally, GÉANT span all over Europe connecting 26 research and educational networks in 30 countries.

The traffic information has been captured in Cisco NetFlow [12] flows format. SURFnet and UT use Cisco system equipment, while the GÉANT relies on Juniper routers that in any case support NetFlow compatible export.

It is already expected that a larger size network such as GÉANT and SURFnet will have a much higher traffic than a smaller one such as UT. As a consequence of that, the processing of all single packets in such large networks is prohibitive due to technological limitations. In order to overcome that, packet sampling is used. The sampling mechanism offered in NetFlow is based on systematic sampling of 1 packet out of *n* (1:*n*). Table 4 shows the sampling rations used by the considered networks as well as the total collected data volume.

Post-collecting phase Once collected the traces, it was necessary to store them in a format suitable for further analysis. In our case, we decided to import flow records into a database [13]. Considering the traces size and the number of flows, we decided to limited our analysis on 2 working days. That corresponds to approximately one fourth of the total collected traffic. So the database cover the period from Aug 1, 2007 00:00:00 UTC until Aug 2, 2007 23:59:59 UTC.

Our MySQL database consists of three different tables, one for each considered network. Each table contains several million of NetFlow records and requires a storage space in the order of tens of GB. In order to speed up the queries, we decided to create indexes on the most used fields of the table. This operation, although needed, resulted to be challenging in both time and space. Importing the traces into a database permits multiple users to access directly the data. Nevertheless, the major limitation to this approach is that extensive queries covering the entire 48 hours period can take hours or even days even

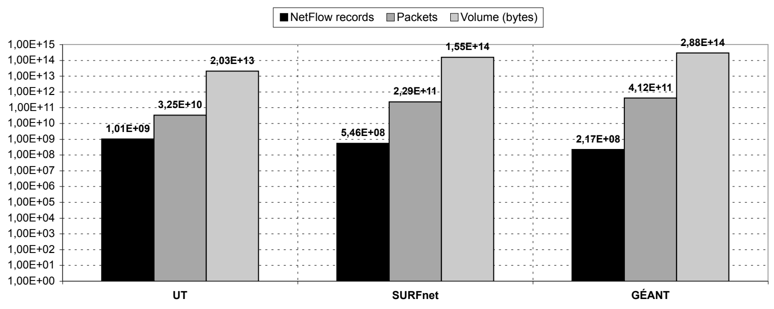


Figure 2: Total amount of traffic per organizational network in logarithmic scale

when the database is entirely dedicated to a single user. All the queries are performed on MySQL 5.0 on EmanicsLab nodes and on 2xIntel Dual-Core Xeon 3.2GHz machines, equipped with 4GB RAM, and running Linux Debian 4.0.

4.1.2 Trace Analysis

In our analysis, different traffic metrics have been considered. First of all, it is important to underline that the entire analysis has been conducted on the basis of NetFlow records.. Some authors proposed different flow definitions according to the traffic information that is considered important [14], while others concentrate their attention to the non trivial problem of estimating the real flow duration, disregarding the exporter timeouts and cache flushings [15] [16]. This topic is beyond the scope of the paper and no flow aggregation has been considered. We conducted our analysis looking at the following metrics: (i) total traffic over the entire 48 hours of analysis period, (ii) level 3 protocols distribution, (iii) flow duration.

For the first two metrics, results are shown according to the number of flows, bytes and packets in the traces. Bytes and packets have been multiplied by the factor of 100 and 1000 in the case of SURFnet and GÉANT, respectively, due to the sampling used in those networks. This provides us with a traffic estimation in those networks.

Traffic overview The first metric considered in our analysis is total amount of traffic. Figure 2 shows the total amount of NetFlow records, packets and bytes per organizational network reported over the analysis period.

Figure 2 also shows that the higher a sample rate is (e.g., UT sampling ratio of 1 out of 1 packet), the bigger the number of flows records will be, which is an expected behavior. One could say, although, that the smaller the sample rate is (e.g., GÉANT sampling ratio of 1 out of 1000 packets), the bigger the chance of selecting a packet not belonging to an existing NetFlow record would be. As a result of that, new NetFlow records could be created in the NetFlow cache, which could increase the number of NetFlow records created. However, Figure 2 shows that this is not the case. It is seen in Figure 2 that the bigger a network scale is, the higher will be the amount of packets and bytes generated.

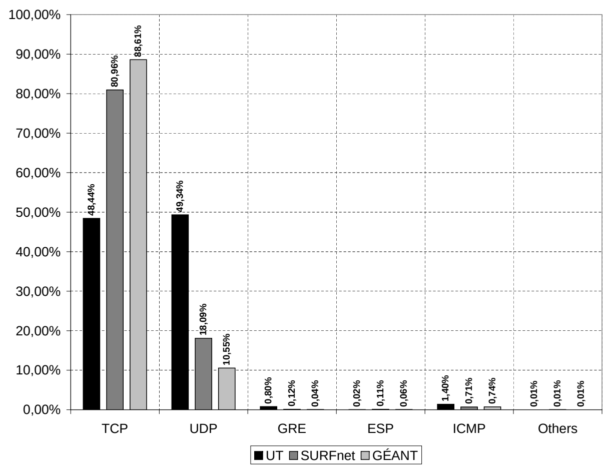


Figure 3: Percentage of NetFlow records per protocol distribution

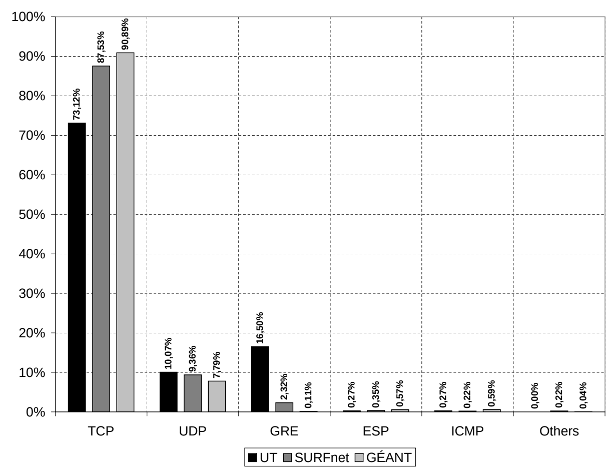


Figure 4: Percentage of packets per protocol distribution

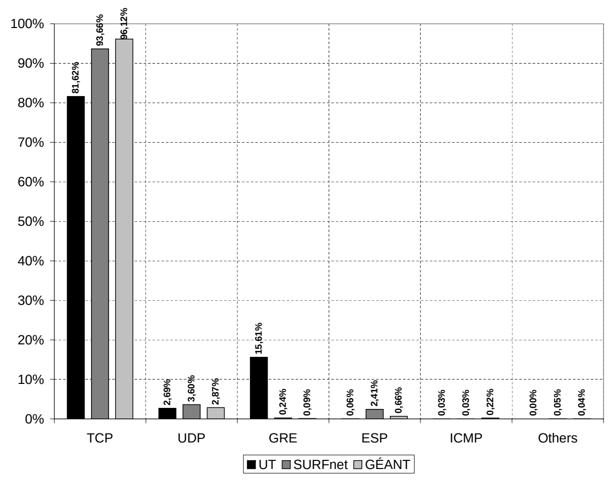


Figure 5: Percentage of octets per protocol distribution

Protocol distribution This subsection shows the protocol distribution observed in the three networks during the analysis period. Five protocols were mostly seen as the Top 5: TCP, UDP, GRE, ESP, and ICMP. Other protocols such as Resource Reservation Protocol (RSVP) did not have significant participation and were combined with the name "*Others*".

Figure 3 shows that most of the NetFlow records have TCP and UDP as protocols. GRE, ESP, and ICMP, and others are a small portion of the total records. Interestingly, the distribution of TCP and UDP records at the UT appear to be different from the ones in SURFnet and GÉANT networks. UT NetFlow records are almost equally distributed between TCP and UDP, with a slight majority of UDP NetFlow records, while SURFnet and GÉANT show mainly TCP ones. The high percentage of UDP NetFlow records may be explained by the fact that the UT is mostly accessed by students who generally use multimedia streams on their daily activities (e.g., IPTV@UT [17]).

Figure 4 shows the percentage of packets per network organization distributed per protocol. It also shows that the percentage of TCP packets is the highest among the protocols in the considered networks. Curiously, GRE packets are more frequent than UDP in the

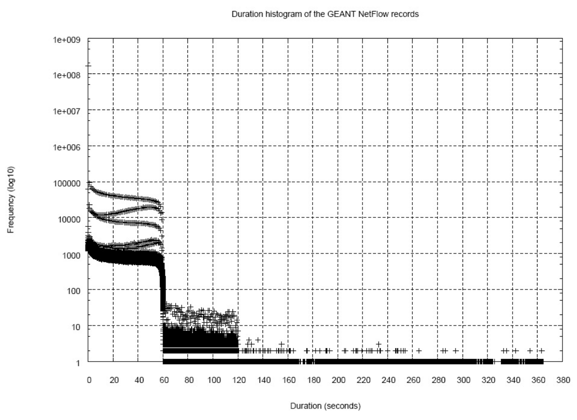


Figure 6: Duration histogram of the GÉANT NetFlow records

UT network. This can mean a constant usage of virtual private networks (VPNs) at the UT domain. GRE packets can also be seen at the SURFNet and GÉANT, but at a discrete amount.

Figure 5 shows the percentage of octets per network organization distributed per protocol. It also shows that the protocol proportion is almost the same presented in Figure 4. This means that most of the traffic volume is consisted of TCP packets whereas UDP packets do not have a significant amount.

Duration distribution This sections presents the duration histograms of the NetFlow records. As told before no flow aggregation has been applied in this work. The flow duration reported here is regarding the duration of the NetFlow record when handled in the NetFlow cache. It is also interesting to mention that we use the same terminology as Cisco to define active and inactive flow timeouts, that is respectively, a timer for long aging flows and timer for flow inactivity.

Figure 6 shows the duration histogram of GÉANT NetFlow records in decimal logarithmic scale (y-axis only). GÉANT uses 5 minutes timeout for active flows and 1 minute for inactive ones. The figure shows that the flow duration frequency drops considerably at 1 minute of flow duration. One explanation for that is due to the sampling ratio used. Small sample ratios increase the chance that a selected packet does not belong to an existing flow entry in the NetFlow cache. As a result of that, NetFlow entries are exported due to their inactivity. One interesting thing to point out here is that NetFlow records can last more than the active timeout as it was also found out at [15]. This can happen in a situation where packets belonging to a certain flow stop being transmitted a bit before its active timeout and retransmitted just before its inactive timeout. This can be seen at Figure 6 where some GÉANT NetFlow records lasted for almost 360 seconds. This allows us to derive the following equation:

$$max\_duration_{flow} \leq active\_timeout_{flow} + inactive\_timeout_{flow} \tag{1}$$

Figures 7 and 8 show the duration histogram of SURFnet and UT NetFlow records, respectively, in decimal logarithmic scale (y-axis only). SURFnet duration histogram shows

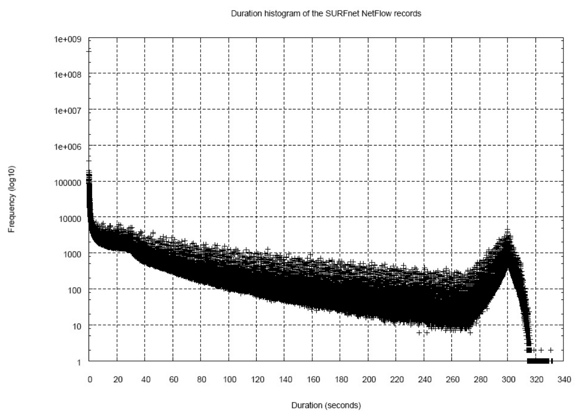


Figure 7: Duration histogram of the SURFnet NetFlow records

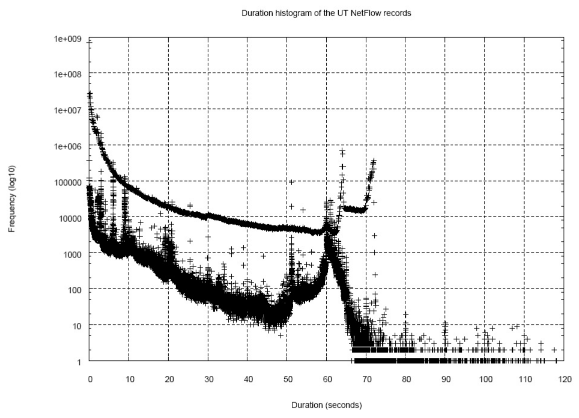


Figure 8: Duration histogram of the UT NetFlow records

in a more visible way the relation between active and inactive flow timeouts. SURFnet uses 5 minutes timeout for active flows and 30 seconds for inactive ones. Looking at Figure 7, it is possible to see that the majority of NetFlow records have a duration shorter than the active timeout and that a pick happens when the active timeout occurs (i.e., at 300 seconds). It is also possible to see that there are some flows that reach the maximum duration, i.e., 330 seconds.

Based on Figure 6 and 7 we can see that the flow duration behavior obeys the maximum NetFlow record duration (Equation 1). However, Figure 8 shows on the other hand an anomalous behavior, which we found out later being a bug in the UT NetFlow-enabled router. UT uses 1 minute as active timeout and 15 seconds for inactive, but Figure 8 shows flows lasting more than the maximum NetFlow record duration. For example, we saw flows lasting for 119 seconds, when the maximum duration expected is 75 seconds. It is also possible to see in Figure 8 two distribution curves, which is most likely a misconfiguration in the UT NetFlow-enabled router. It is apparently reporting duplicated duration times for the NetFlow records.

An interesting fact found in our analysis was the amount of flows with duration equal to zero reported in the three considered networks (Figure 9). The usage of sampling explains why most of the NetFlow records are reported with duration equal to zero. However, in the

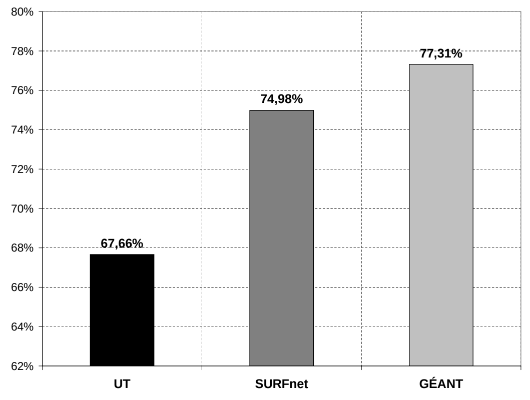


Figure 9: Percentage of NetFlow records with duration equal to zero

case of UT, where sampling was not used, the percentage of flows with duration equal to zero is considerable high. At the moment, we do not know yet the reason why so many UT NetFlow records have duration equal to zero. It may be related to the bug found out in the UT Cisco router. This issue is still under investigation though.

4.1.3 Conclusions

In the paper, we presented a flow-based traffic characterization on traces collected on different network scales (UT, SURFnet and GÉANT) and with different sampling ratios.

The operational experience we have gained during the data acquisition process (collecting and post-processing phases) made clear that measurement and analysis of such extensive quantity of data coming from different locations is not a trivial operation. In particular, all the involved steps required large resources: disk space for the initial collection, memory and CPU power for data normalization, data-base importing and analysis. Above all, the most challenging constraint is time: importing, data-base indexes creation and query processing are not only CPU-consuming operations, but they also requires hours and sometimes days to be completed.

Moreover, the implementation of the collecting and analysis infrastructure can be only made on the basis of design choices that in some cases can be defined as critical for the further research: the storing formats, the data-base type and the information to be imported. Regarding the number of entry in the database, also data-warehousing solution could be considered in order to speed up the analysis process.

Regarding our analysis, we can conclude that the traffic shows, in almost all cases, the same characteristic in the three networks and that the considered metrics scale accordingly: flow records, packets and bytes have the same growing trend and TCP and UDP are the most common protocols and all the network show. Moreover, at least two of the considered networks show a common behavior regarding the duration distribution.

In some occasion we noted exceptional behaviors, e.g. the UDP/TCP flows repartition in the UT trace, or the large use of the GRE protocol in the same network. Some of these anomalies can be explained taking into account the peculiarity of the network. Nevertheless, others, in particular the large number of flow records with duration zero at UT, are still

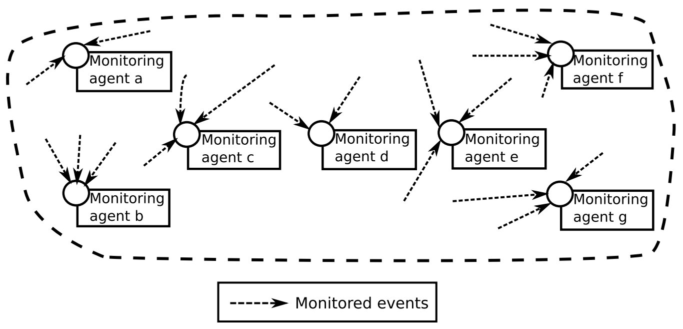


Figure 10: Distributed monitoring model

unsolved: sampling explains the bias towards flow with duration equal to zero in the case of SURFnet and GEANT. The absence of sampling at the UT should result in a different behavior, but this does not happen. The issue is still subject of investigation.

4.2 Activity monitoring for large honeynets and network telescopes

The motivations of this work are twofolds. The first motivation is related to the conceptual approaches and algorithms required to perform distributed monitoring. If we consider a distributed monitoring platform for a given target deployment (please see figure 10), several questions must be addressed.

- Do all management agents observe the same type of events? If no, how can we correlate a distributed view and aggregate the commonly observed evidence?
- Can we discover a temporal behavior of the whole platform? Do some agents tend to observe the same type of behavior during a particular time of the day, while others remain to hold a localized and very isolated observation behavior?

A second motivation came from a very realistic requirements. We are part of a large honeynet distributed over the Internet. Each individual honeypot monitors backscatter packets and incoming attacks. When working on the resulted datasets, we were challenged by the lack of methods capable to compare such a distributed platforms and to detect temporal/spatial trends in the observed traffic patterns. In our work we had to process similar attack traffic from a different security monitoring platform (a network telecope) and compare it to the results obtained from the honeynet.

This section is structured as follows: In section 4.2.1, a generic method for analyzing a distributed monitoring platform is described. This method uses graph intersections in order to model the distributed platform and to follow their temporal evolution. Section 4.2.2 and 4.2.3 show how this method can be used for two realistic distributed environments : a honeynet and a network telescope. An analysis concerning IP related headers is done for the two data sources and additional results concerning differences and analogous behavior between these two are presented. Section 4.2.4 concludes the section.

4.2.1 Intersection graphs

The method based on intersection graphs has been introduced in [18] for profiling communications patterns between the users of a high profiled enterprise.

Graph A graph is composed of several nodes and arcs. Two nodes are linked if there is a relation between them. A relation can be: similarity, difference, or communication exchanges. The relation will be formally defined for each deployment target in the following sections. We consider that arcs are not directed and that the graph is an undirected graph. The adjacency matrix of a graph is a boolean square matrix where each line and each column represents a node. It is defined as :

$$\begin{aligned} A_{ij} = \quad & 1,\ if\ an\ arc\ between\ i\ and\ j\ exists, \\ & 0,\ else, where\ i\ and\ j\ are\ 2\ vertices\ of\ the\ graph \end{aligned} \qquad (2)$$

Since we consider a undirected graph :

$$A_{ij} = A_{ji}(symmetrical\ matrix)$$

Central node Generally, a central node is interesting because it has multiple direct or indirect relations. Using the most central node we can evaluate the centrality of the graph by counting the number of relations (arcs). A simple method to detect this node could be to get the node which has the maximum number of neighbors. However, this simple method ignores nodes that have only few relations but these relations lead to nodes that are well connected. Therefore, we can consider not only the direct neighbors but a subgraph of all nodes which are located in an area defined by the distance from the evaluated node. The *centrality* is the number of arcs of the subgraph. Another way to get the central nodes is to use the eigenvalues and eigenvectors, as proposed in [19].

Locality statistics A graph can vary over the time and thus we need to somehow capture and describe variations in the centrality. The main idea is to consider at each time instant the central node and the associated centrality and to analyse the temporal behavior of these two entities. The intuition behind is that when major graph changes occur in the topologies of a graph, the relations between nodes change and this will be reflected by a change in the centrality too. Moreover, the central node which is responsible of the maximal centrality can be detected and the appearance or disappearance of a node implies that its relationships increased or respectively decreased. The following formula describes formally the locality statistics, described in the previous paragraph :

$$\psi_k(v) = \ number\ of\ arcs\ of\ the\ subgraph$$

$$of\ k\ nearest\ neighbors\ of\ v$$

$$M_k = \max_{v \in nodes} \psi_k(v) \qquad (3)$$

The value of k has to be chosen carefully. For k = 0, the value is always 0 which is normal because in this case no neighbors are concerned and only the current node composes the subgraph. k must not be too small because important information might not be revealed. If k is to large, all the graph is covered.

The major goal is not only to show the evolution of the topology of the graph but in fact to discover new nodes that might become important, even if they are not central, i.e., their centrality is lower than the centrality of the central node. For this, the standardized locality statistics is useful:

$$\tilde{\psi}_{k,t}(v) = \frac{(\psi_{k,t}(v) - \hat{\mu}_{k,t,\tau}(v))}{\max(\hat{\sigma}_{k,t,\tau}(v), 1)}$$

$$\hat{\mu}_{k,t,\tau}(v) = \frac{1}{\tau} * \sum_{t'=t-\tau}^{t-1} \psi_{k,t'}(v)$$

$$\hat{\sigma}_{k,t,\tau}(v) = \frac{1}{\tau - 1} \sum_{t'=t-\tau}^{t-1} (\psi_{k,t'}(v) - \hat{\mu}_{k,t,\tau}(v))^2$$

$$\tilde{M}_{k,t} = \max_{v \in nodes} \tilde{\psi}_{k,t}(v) \qquad (4)$$

In fact, in the formula 4, the centrality is standardized with respect to previous values of a sliding window. The size of the window is τ. Nodes which tend to remain constant will have a low value.

From graphs to network monitoring If we consider a distributed monitoring platform, we can use a graph model to represent the relationships among the monitoring agents. Each agent is represented by a node in the graph. A relationship between two agents is given by the similarity in the observed data and is thus domain specific. The major idea however is to consider an arc between two nodes, if and only if the associated agents have observed a common activity. To illustrate this idea, if we consider different honeypots of a honeynet and each honeypot monitors commonly used parameters like source IP addresses, source ports, destination ports, an arc between two nodes exists if both agents have a significant overlap in the observed parameters.

4.2.2 Honeynet and intersection graphs

A honeypot is described in [20] as an environment where vulnerabilities are deliberately introduced. Malicious intruders are lured into attacking such a system and providing useful information to security officers and researchers. Such information typically includes details about the source of the attack, temporal patterns in this activity and the tools used during and after an attack. However, only one honeypot is not sufficient for a sound analysis at a Internet scale level. Several honeypots can be grouped into a network which is called an honeynet. In this case, all honeypot share their informations with others and they are dispersed over all the Internet.

For our work, the honeynet of the Leurre.com project was used. This network consists of 129 individual systems run by 43 honeypots. Each individual honeypot uses 3 distinct IP addresses and emulates 3 different operating systems (one operating system per address : Windows NT server, Windows 98, and Linux Red Hat 7.3). Data is collected locally and centralized in a database. The period of our study covers the data from May to December

		Honeypot
#monitored addresses		129
Number of incoming packets	05	475 519
	06	1 211 820
	07	1 495 525
	08	1 821 534
	09	1 371 280
	10	2 317 525
	11	2 292 083
	12	1 451 770
Number of unique source IP addresses	05	18 392
	06	39 419
	07	34 011
	08	49 076
	09	60 666
	10	77 032
	11	84 485
	12	82 500
Size of data	05	69 MB
	06	176 MB
	07	217 MB
	08	264 MB
	09	199 MB
	10	337 MB
	11	333 MB
	12	211 MB

Table 5: Global information about the honeynet data. The months are represented in number (05, 06, 07...)

2004 and includes more than 11 millions IP packets. The period is sliced into weeks. The table 5 gives the exact details about the analyzed data.

Source IP addresses The goal of our first analysis was to analyse the distributed views of the honeypots with respect to the source IP addresses and identify the ones that stand out of the crowd, ie that capture suspect source addresses that are not captured by other honeypots.

Nodes represent the different honeypot platforms. For each nodes, the sets with captured source addresses are compared. Two nodes are linked only if the intersection between the corresponding sets represents less than a threshold of the union of addresses. If nodes were really distinct, there would be more and more arcs and the locality statistic would increase. The normalized locality statistic permits to detect where and when the topology changes significantly and to detect the honeypots which are responsible for the new maximal locality. These central honeypots could be considered as interesting because they detects particular source IP addresses

Figure 12 shows the plots of the simple locality statistics, using several threshold percent-

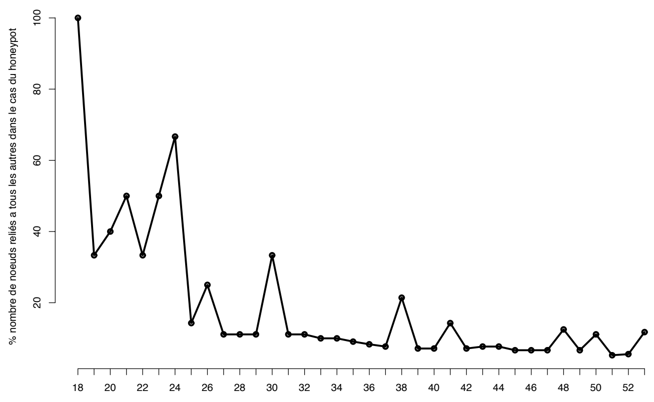


Figure 11: Number of central nodes for the honeynet

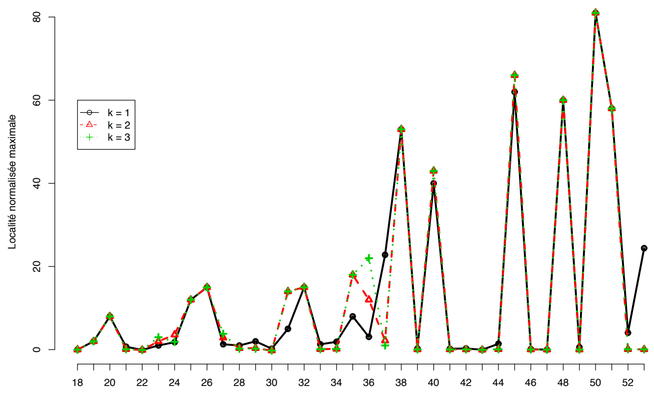


Figure 12: Honeynet source IP addresses analysis - Standardized locality with $\tau = 5$ (shared addresses $\leq$ 0.25%)

ages. For small thresholds, the plots tend to overlap, a good setting of this value is 0.25%, where only few points are not overlapped.

On the average, there are one or more nodes having high centrality values, because these nodes are linked to all other nodes. The figure 11 shows the number of this type of nodes as well as the respective honeypot. The first value is less important because in this case all nodes are isolated. The number of honeypots that are significantly different (0.25%) decreases too. The figure 12 represents the standardized locality with $\tau = 5$ weeks. Using the method of the intersection graph, we can observe that when the value of the maximum standardized locality statistics is low, the topology of the graph is constant, while high values indicate major topology changes. The plots are generally overlapping and there are 8 peaks. The concerning central nodes have been extracted and some nodes (6) appear several time. Therefore, the 6 honeypots corresponding to these nodes are very different with respect to the remaining ones.

Source ports A second goal was to detect honeypots which observe port source addresses that other honeypots have not observed. Only packets with both flags SYN and ACK were considered. This kind of packets are in fact backscatter packets (a more de-

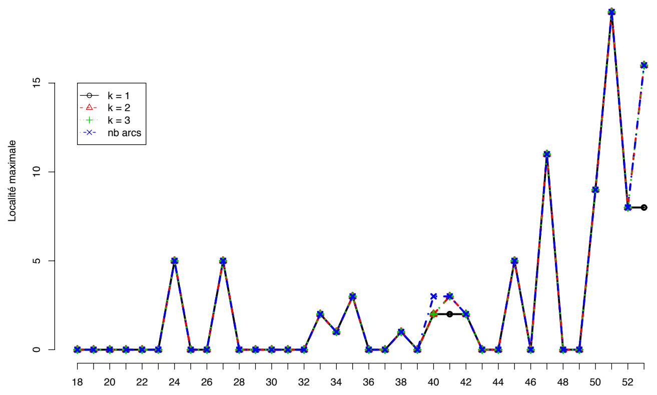


Figure 13: Honeynet source ports analysis - locality statistic (shared ports $\leq$ 10%)

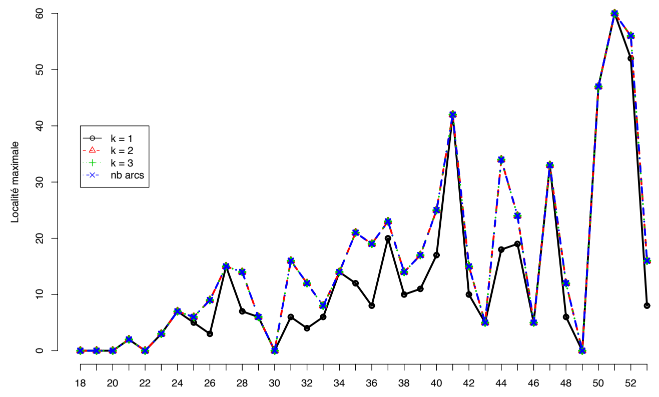


Figure 14: Honeynet source ports analysis - locality statistic (shared ports $\leq$ 25%)

tailed overview on backscatter is given in section 4.2.3). In this particular case, the perceived source ports are in fact ports which have been attacked with IP spoofed packets. Thus, this study is relevant to attacked ports.

A node in the graph is a honeypot platform and similar to the previous case, an arc links 2 nodes if the set intersection of their source ports is lower than a threshold of the union of the source ports. Therefore, if honeynets were different, the locality statistic of these nodes would increase and the plots of the maximal locality statistic would show it. The plots corresponding to the unnormalized maximal locality statistic are represented in figure 13 (for a threshold of 10%) and respectively in figure 14 for a threshold of 25%. A threshold of 25% implies that the number of arcs is higher and the different plots are not overlapping. However, the aim of our work was to detect platforms that are different and a 25% threshold means that we consider 2 honeypots different even if they share one quarter of their source ports. If we consider both thresholds 10% and 25% we observe that the peaks in both plots are located at the same time instants and such the threshold of 10% is sufficient for detecting topology changes. The plots of the maximal centralized locality statistic with a sliding window size of 5 look like the figure 13 and 14.

If we consider now the plots for a threshold of 10%, at many time instants the number of arcs is 0. In these cases the honeypots share more than 10% of the detected attacked ports. The ports are coded with 2 bytes in the TCP header and so 2^{16} ports are theoreti-

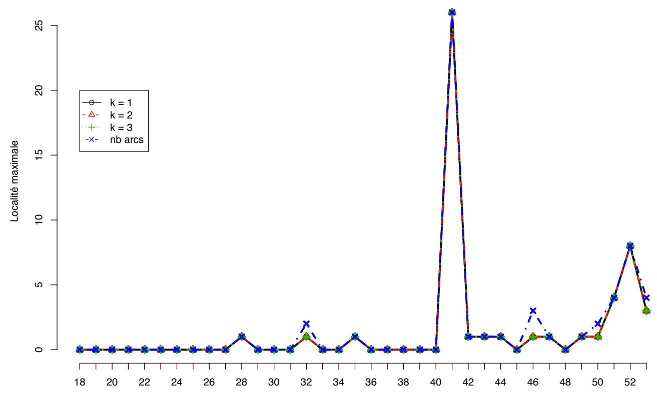


Figure 15: Honeynet acknowledgment numbers analysis - locality statistic (shared acknowledgment numbers $\geq$ 90%)

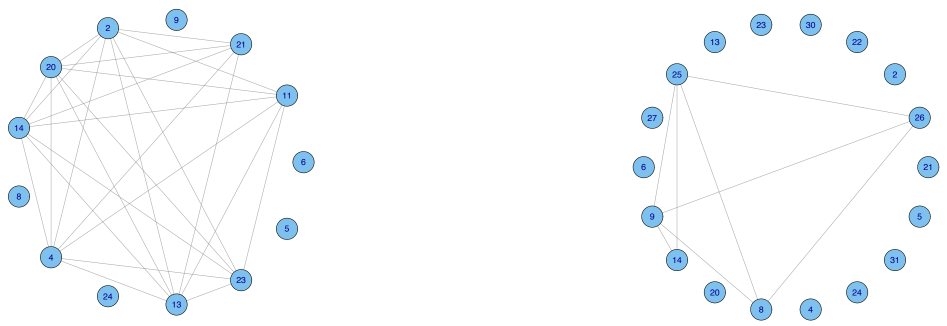
cally possible. However only few ports out of this large pool are really used and correspond to known deployed services.

Although several peaks are visible, the maximum locality is not very high and it's probably due to the low quantity of data at the honeynet. For instance,if the ports detected would be completely different between the 43 honeypots, the number of arcs would be : $\sum_{i=43-1}^{1} i = 946$.

Attack tools A TCP session is established thanks to the 3-way handshake. First the initiator sends a packet with flag SYN and a random sequence number (also called Initial Sequence Number -ISN). The correspondent acknowledges the packets with an acknowledgment number equal to the previous sequence number + 1. Finally the initiator acknowledges this reply. Some attack tools use always the same sequence number or do not use a good (high entropy) random number generator. Consequently, the acknowledgment numbers are either always the same, or depend on the use of a specific exploit code. We looked if the same attack tool was used to attack different computers and for this work we considered also the the backscatter packets (replies of attacks).

In this case, the construction of the graphs consists in considering nodes as honeypots and two nodes will be linked if they share more than a threshold of the union of their observed acknowledgment numbers. Using a threshold of 90% the plots are given in figure 15. In general the acknowledgment numbers are different between platforms because the number of arcs is low. This is due to the diversification of the attack tools.

Two peaks are clearly visible and in these case the plots are overlapping. This shows the presence of one or central honeypot linked with all others. Using the standardized locality statistic with a sliding window size of 5, the obtained plots are similar because the standardization is made thanks to previous values, which are mostly equal to 0. The figure 16 presents the graphs of weeks 41 and 52 corresponding to the peaks. In the figure 16(a), many nodes are linked with many others. A lot of honeypots have detected about the same acknowledgment numbers (threshold $\geq$ 90%) and the use of the same attack tools is undeniable. However for the second peak in week 52, (shown in the figure 16(b)) the picture is totally different and only some honeypots are concerned. In this case, this is probably



(a) week 41 (b) week 52

Figure 16: Intersection graphs for acknowledgment numbers shown by the honeynet

due to a same attack tool with a bad random numbers generator which implies that the same generated number is used several times and detected by different honeypots.

4.2.3 Network telescope and intersection graphs

The principle of network telescope is simple. A monitoring devices saves all incoming traffic to a specific range of IP addresses. In fact, these addresses are unused and cover a range which is generally a subnetwork of consecutive addresses. The main characteristic of a telecope is its size which is generally huge. It is possible to create more interactive network telescope which emulate diversified services like shown in [21].

We used in our work data from the telescope developed in the CAIDA project [22]. The monitored addresses form an A class network and the number of addresses is 2^{24}. This number outweights the number of honeypots in the honeynet which used only $3 * 43$ addresses. This huge telescope gathers data from a fraction of $\frac{1}{256}$ of the Internet. Only backscatter packets are captured by this telescope. Backscatter packets are generated indirectly by a denial of service attacks and for a comprehensive overview, the reader is referred to [23]. The basic scenario is as follows: an attacker does a SYN flooding of a victim in order to force the victim to reply to each packet. The attacker can spoof the source IP addresses in order to hide her identity and avoid additional bandwidth consumption on her side. The victim of the denial of service attack replies to the spoofed addresses and these replies are called backscatter packets. The figure 17 shows a simple scenario where an attacker spoofs three IP addresses but only one is assigned to a real and legitimate network interface. The others are a part of the addresses of a telescope which collect these backscatter packets.

During our analysis, only the period from 26 to 36 August 2004 is studied on a hour by hour basis. About 460 millions of packets have been gathered during this period corresponding to 24.1 GB of data. For more information about the data, please refer to the table 6.

Source IP addresses The goal of this study is similar to the previous honeynet analysis. We wanted to detect if a part of a telescope detects source IP addresses which are not detected by other parts. The range of IP addresses monitored is sliced into several /16

		Network Telescope
#Observed IP source addresses		116 777 216
Number of incoming packets	2004/08/26	52 784 835
	2004/08/27	88 411 307
	2004/08/28	142 096 855
	2004/08/29	77 094 947
	2004/08/30	51 850 438
	2004/08/31	45 742 568
Number of unique source IP addresses	2004/08/26	171 257
	2004/08/27	244 643
	2004/08/28	241 883
	2004/08/29	242 491
	2004/08/30	231060
	2004/08/31	246 982
Size of data	2004/08/26	3,8 MB
	2004/08/27	6,3 GB
	2004/08/28	1,5 GB
	2004/08/29	5,5 GB
	2004/08/30	3,7 GB
	2004/08/31	3,3 GB

Table 6: Global information about the telescope data

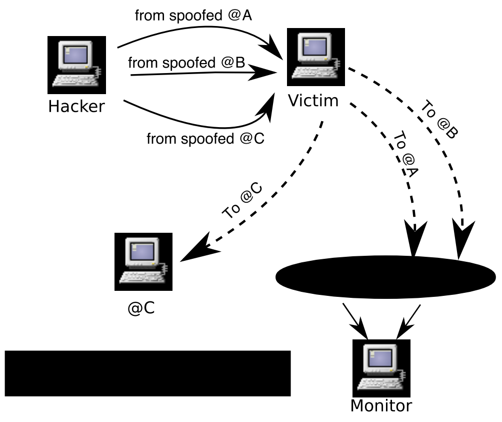


Figure 17: Backscatter principle

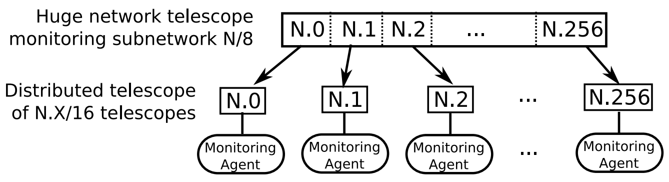


Figure 18: A distributed telescope

subnetworks. Because of the size of the telescope is a /8, we consider $2^8 = 256$ subnetworks. This division is logically equivalent to a distributed monitoring model described in figure 10. When this model is instantiated, we obtain the architecture illustrated in figure 18. In fact, each subnetwork of the telescope is considered as an entity for which there is one monitoring agent.

The nodes are the subnetworks and two nodes are linked if the intersection of their source IP addresses is less than a threshold of their union. If the subnetworks were really different in term of observed source IP addresses, a lot of links would appear and the locality statistic would increase.

We have tested threshold values of 5% and the maximum locality statistic is always 0 except for the first hour which is probably due to a lack of data at the beginning of the capture (because the August 26 is the first day of August for which we have data). A threshold value of 5% is low but we also intended to compare honeynet and telescopes and we concluded that there is a high redundancy of information in the telescope case.

Source ports The packets that have been captured by the telescope are only backscatter packets and so the source ports of these packets are in fact attacked ports. It's interesting to study them in the same manner that we have done it for the honeynets. The difference here is that the nodes are the subnetworks of size /16 of the range of monitored IP addresses. Our goal was to detect if sometimes, only particular ports were attacked.

Using a threshold of 5% we obtained the plots shown in figure 19. The number of arcs and the locality statistic is close to 0. The source ports shown by the different subnetworks are the same. The conclusion is the same as for the honeynet case : attackers attack frequently the same ports and the telescope can detect this phenomena.

A peak appears clearly on the figure 19 and in fact there are 3 subnetworks detecting unusual source ports. This is opposed to the honeynet case for which a peak is not always significant due to a low amount of data. Because a telescope monitors a fraction of $\frac{1}{256}$ of the Internet, a high peak like its shows a real specific phenomena at this time and this peak is a proof of attacks on original ports.

4.2.4 Conclusions

In this work we were challenged by several research questions. Firstly, we needed a generic method to analyze both telescope and honeynet data. The main goal was to compare these two ways of gathering malicious network trafic. While a telescope monitors

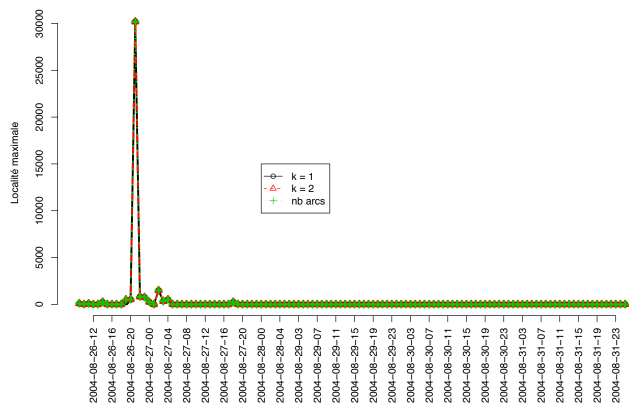


Figure 19: Telescope source ports analysis - locality statistic (shared ports $\leq$ 5%)

a large range of consecutive IP addresses, the honeynet monitors a limited set of IP addresses dispersed over the Internet. The amount of data is much higher for the telescope if compared to the honeyet. A second contribution of our work was to assess the utility of each method to collect network information. For instance, we have observed that a honeynet is sufficient for learning the distribution of source addresses, contrary to telescope for which a high redundancy might become an obstacle in the analysis.

On the other hand, both methods did provide similar results about the services/ports that are atacked, but the telescope is superior when detecting less frequently attacked services. This is quite obvious, due to the much higher data volume. Concerning the used attack tools, the honeynet permitted to show that these are more and more diversified and sophisticated.

The central concept underlying our work are the intersection graphs. These graphs have not been used widely in the field of network security. The advantage of this method is that analyzing aggregated data is possible by considering the curve of the maximum locality statistic and the maximum standardized locality statistics. This is possible because these plots are closely related to the trend of the variation in the topology of a graph. This method allows also to identify the nodes, which are important in the graph. Importance can be assimilated with monitoring agents that observe unusual network activities.

Several papers individually analyzed either telescope data or honeynet data, but none had tried yet to compare these two data source simultaneously. Our work is to the best of our knowledge the first attempt to compare the two methods over the same time period.

4.3 Detection of Botnets using Flow-based Prefilters

As described above, Flow-based Network Intrusion Detection (FNID) Systems attempt to reduce the amount of data to process by operating on abstracted summaries of network traffic. As the abstraction process reduces the amount of information available about activities, while every detail may prove crucial in complex attack scenarios, this step often results in the reduced alert confidence mentioned (e.g.) in [24].

As Flow information is solely assembled on the basis of packet information, it not always obvious how FNID can lead to an improvement in detection rates over traditional Packet-based Network Intrusion (PNID) concepts. Even more so, the reduced amount of information furthermore renders labelling of detected attacks more difficult due to the poor amount of details.

Consider a classification of attacks w.r.t. alert confidence, where attacks may either be detected reliably[1], unreliably[2], or not at all. In an ideal world, where Intrusion Detection is not constrained by any resource limitations and all required information is available, this would yield a situation, where no additional attacks could be detected by FNID concepts w.r.t. PNID concepts at all. This situation is displayed in Figure 20 on the left hand side. Therein, PNID detection performace is determined solely on the amount and correctness of information available about attack properties required for detection.

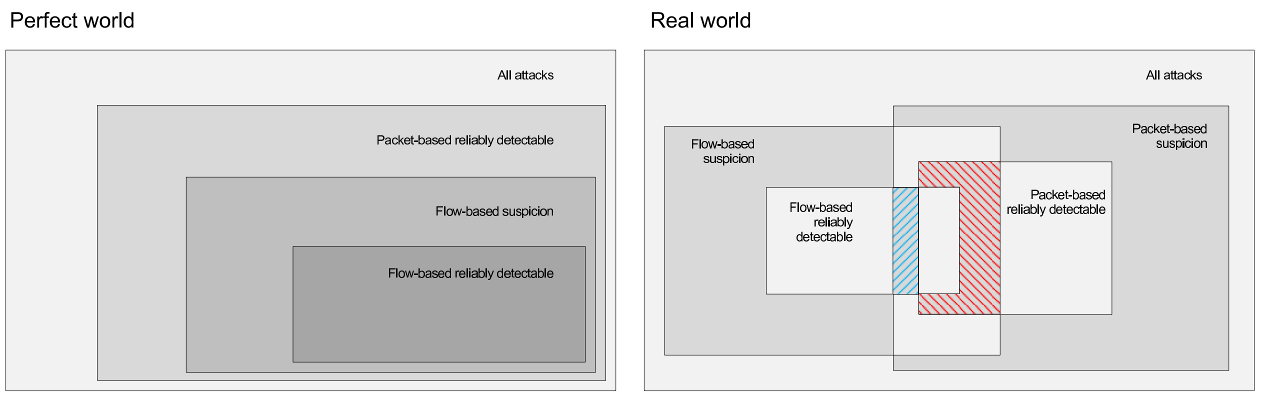

Perfect world
All attacks
Packet-based reliably detectable
Flow-based suspicion
Flow-based reliably detectable
Real world
All attacks
Packet-based suspicion
Flow-based suspicion
Flow-based reliably detectable
Packet-based reliably detectable

Figure 20: Set theory model of concepts

However, as the real world imposes resource and informational availability constraints, PNID detection performance is also influenced by the completeness of information about network activities, which may be reduced by failure to inspect the entire traffic. This results in a situation displayed on the right hand side in Figure 20, where FNID may be able to detect attacks not reliably detectable by PNID.

In the context of SNID, is is important to evaluate, whether a hybrid model of FNID concepts and PNID concepts would allow to profit from FNID's scalability, while compensating for abovementioned reduction in alert confidence. The most obvious cooperation angles are given in situations, where suspicions may be raised by use of the FNID concept, which may be verified and correctly labelled using the PNID concept.

This hybrid model uses flow-based concepts as prefilter for traffic to be inspected on a packet level. In the context of Botnets, we isolate traffic, which is suspected Botnet communication, on flow level and verify on a packet level, whether the suspicion holds or whether it doesn't.

This section will present the evaluation environments chosen, elaborate on mechanisms considered for PNID and FNID hybrid detection, and describe the scenarios, based on which we intend to perform a resource and detection efficiency evaluation.

[1]i.e. with a low amount of False Posititives and False Negatives
[2]i.e. with either a high amount of False Positives or a high amount of False Negatives

4.3.1 Evaluation Environment

As the resource efficiency gain[3] of abovementioned hybrid model can be expected to depend heavily on the reduction of traffic inspected by the PNID System, it is likely not only to depend on the quality of the flow-based prefilter, but also on the ratio between legitimate traffic and actual attack traffic.

As realism is hard to guarantee in synthetic traces, and as the few traces available ([25],[26]) are either too old to contain botnet traffic, or are unclear about what attacks they contain, we decided to choose real time traffic for evaluation. As the evaluation requires large amounts of data, rendering storage and playback not an option, live traffic will be the only option.

The evaluation will be performed in three separate environments:

- CSG Testbed
- Internet Uplink University of Zurich
- Swisscom DSL backbone

CSG Testbed The Communication Systems Group (CSG) at Department of Informatics (IFI) at University of Zurich (UZH) maintains a testbed of 30 server nodes dedicated to research purposes. In order to guarantee unblocked Internet access (by firewalls, NATs and other potential filters), the connectivity is entirely independent of the university link. The connection from the Swiss Research Network SWITCH is split at the first switching point of the university and directly connected to the testbed via a dedicated fibre-optic link. The testbed carries two so called PlanetLab nodes [27] that are mainly used for testing peer-to-peer applications and therefore generate a lot of such traffic. Even though internet connectivity is 1 Gbit/sec rates above 20 Mbit/sec can be observed only rarely.

The CSG testbed is used as the primary development environment for our implementation, evaluation was limited to sanity checking of policy scripts. However, the bandwidth and packet rate are too low in order for the tests to deliver useful results to load-testing.

The testbed monitoring machine is a Dell PowerEdge 850 with a Pentium 4 CPU running at 3.6 GHz. It is equipped with 3 GB of RAM and running Ubuntu Linux 7.04 (Feisty).

More information of the testbed setup can be found at [28].

Internet Uplink University of Zurich Our first primary evaluation location is located at the Internet uplink of the University of Zurich. A dedicated server for monitoring purposes was already installed by the responsible security chief of the university network. The university network is connected to the Internet via the Swiss Research Network SWITCH by a 10 gigabit Ethernet link. Despite the high connection speed, the link is never used for more than a couple percent. This allows packet capturing using standard gigabit Ethernet network cards.

[3]with respect to simple PNID

The monitoring server uses two network interfaces for capturing traffic outside (network interface *eth0*) and inside the firewall (network interface *eth1*). The amount of traffic seen outside the firewall is larger than on the interface on the inside as it contains connection attempts to hosts protected by the firewall. The monitoring server contains four Intel Xeon processors running at 2.4GHz and 4 GB of memory.

As we are not able to use the monitoring server exclusively, the impact of other monitoring processes will be minimized by setting their affinity to only one core while our measurement processes will be allocated on the remaining three cores. It was planned to use the alternative PF_RING [REF] library for the evaluation. However, due to severe instability caused by the installation, we were forced to fall-back to the standard pcap library.

The university firewall is configured to allow all traffic from the university network to the Internet and generally blocks all traffic from the outside. An exception is port 113/TCP, normally used for the *identd* service in IRC. As a consequence, functionality of peer-to-peer applications is generally limited, as incoming connections are blocked by the firewall. Some users know that port 113/TCP is not blocked and configure their peer-to-peer application to use this port for incoming connections accordingly. Malicious peer-to-peer applications often fail to find this open port and instead rely on establishing sessions from the inside in order to retrieve commands.

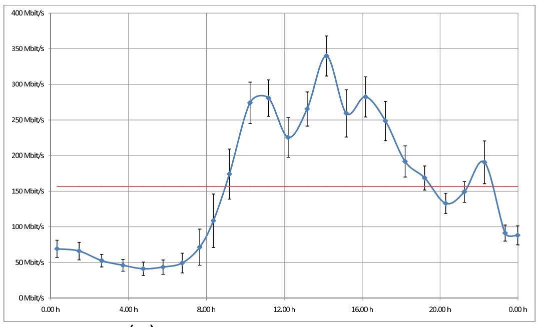
All hosts in the university network have public IP addresses in the range 130.60.0.0/16. NAT is not used.

In order to characterize the link in terms of intrusion detection relevant performance parameters, hourly samples of 10 million packets were taken with *tcpdump* on 24.01.2008. The recorded samples were then processed with the *tcpdstat* software to determine throughput in megabits per second, packet frequency (number of packets per second) and the packet size distribution. Figure 21(a), 21(b) and 21(c) show the results of these measurements.

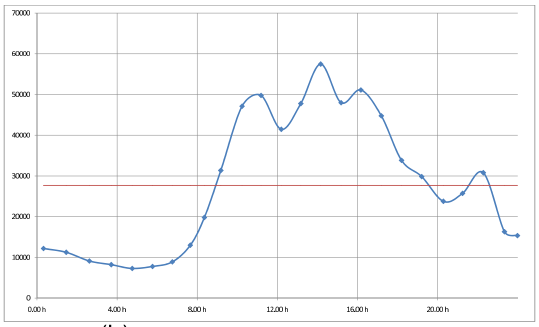
Figure 21(a) shows the hourly throughput measurement in megabits per second. The average (red line) lies at 156.5 Mbit/s, the peak at 14:09h at almost 340 Mbit/s. The standard deviation shown by the vertical indicators at the measurement points, averages at 20.75 Mbit/s. Deviations during a single measurement can occur because the sample measurement is terminated by the number of packets, hence throughput can change during the measurement itself.

Figure 21(b) shows the packets frequency measurement in packets per second. The curve largely correlates with the throughput measurement. The average lies at 27670 packets/s (red line), the peak at 14:09h at 57460 packets/s. Figure 21(c) shows the packet size distribution in bytes of the combined sample measurements throughout the day. 43% of the packets are smaller than 128 bytes, 45% are larger than 1023 bytes, 12% have a size between 128 and 1023 bytes. The MTU for most hosts is observed at 1434 bytes with 29% of all packets having this size and only 0.58% with a size larger.

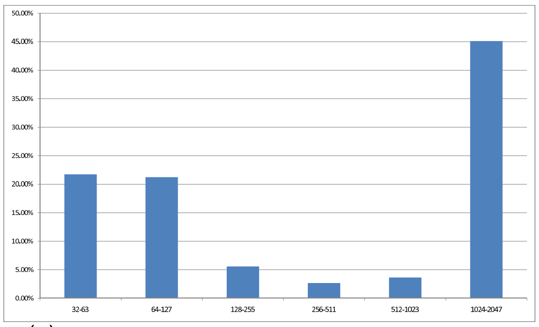
Swisscom DSL backbone The second evaluation site is located at the Swisscom's network site in Zurich-Binz. Swisscom operates as the largest Internet service provider in Switzerland under their "bluewin" brand. The monitored link has a significant higher throughput and packet frequency as our other evaluation sites. Two ports were mirrored



(a) Throughput measurement



(b) Packet frequency measurement



(c) Packet size distribution measurement (in bytes)

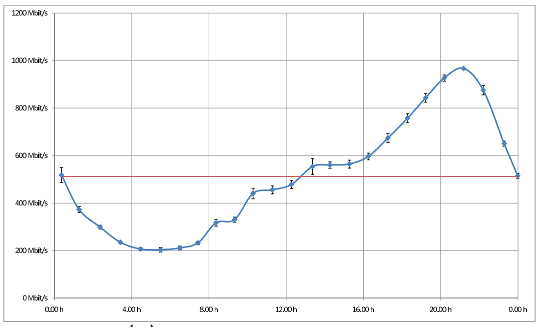
Figure 21: University of Zurich Internet Uplink, 24.01.2008

from a load-balancer and connected to our own, exclusive monitoring server using two gigabit Ethernet links. These links were highly utilized and carry traffic from some thousand mostly private ADSL and VDSL customers. Naturally, many peer-to-peer connections can be observed, as many users use this type of service from their home Internet link. No firewalls or other filtering are present on the provider side. The traffic is genuine unfiltered Internet traffic of hosts that are the primary botnet and worm infection targets and therefore provides an ideal evaluation environment for our implementation.

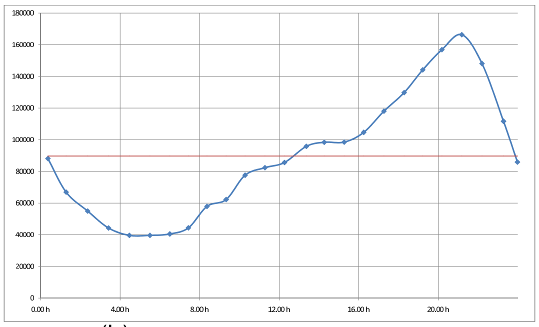
The alternative pcap library PF_RING was installed, as the amount of packets that have to be would not be processable with the standard pcap library. Our monitoring server uses a Intel Dual-Core Xeon processor at 2.66GHz and is equipped with 8 GB of memory. Packet capturing on the two network interfaces simultaneously is theoretically possible using the virtual *any* network interface but comes with drawbacks such as missing packet filtering statistics. It was therefore decided to perform the evaluation only on one of the provided interfaces carrying more traffic (*eth1* with peak throughput of up to 960 Mbit/sec to *eth0* with 350 Mbit/sec).

The alternative pcap library PF_RING was installed, as the amount of packets that have to be would not be processable with the standard pcap library. Our monitoring server uses a Intel Dual-Core Xeon processor at 2.66GHz and is equipped with 8 GB of memory. Packet capturing on the two network interfaces simultaneously is theoretically possible using the virtual *any* network interface but comes with drawbacks such as missing packet filtering statistics. It was therefore decided to perform the evaluation only on one of the provided interfaces carrying more traffic (*eth1* with peak throughput of up to 960 Mbit/sec to *eth0* with 350 Mbit/sec).

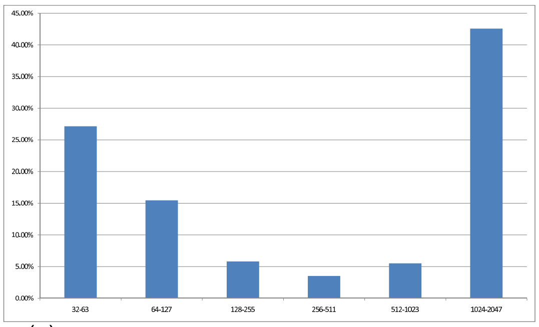
During peak periods up to one third of CPU time was used for interrupts generated by the



(a) Throughput measurement



(b) Packet frequency measurement



(c) Packet size distribution measurement (in bytes)

Figure 22: Swisscom DSL backbone, 05.02.2008

network interface cards.

Equivalent to the University link, we have taken hourly packet samples throughout a weekday and processed them with *tcpdstat* in order to characterize the link in terms of intrusion detection relevant network parameters.

Figure 22(a) shows the hourly throughput measurement in megabits per second. The average (red line) lies at 511 Mbit/s, the peak at 21:11h at 967 Mbit/s. Average standard deviation is 14.4 Mbit/s. The curve is clearly different from the University measurement. The simple explanation for this is that the Swisscom site carries more traffic of home users that are particularly active is the evening compared to the University site where most traffic is observed during working hours.

Figure 22(b) shows the packet frequency measurement in packets per second. As for the University site, the curve largely correlates with the throughput measurement. The average lies at 89689 packets/s (red line), the peak at 21:11h at 166334 packets/s. The average load is therefore about three times higher compared to the University site.

Figure 22(c) shows the packet size distribution in bytes of the combined sample measurements throughout the day. 42,5% of the packets are smaller than 128 bytes, 42,5% are larger than 1023 bytes, 15% have a size between 128 and 1023 bytes. The distribution is roughly the same as at the University site. An MTU value for most hosts could not be determined, as traffic from various sources seem to have different limitations.

4.3.2 Evaluation of Mechanisms

As our focus lies in the intersection of the “flow-based suspicion” and “packet-based reliably detectable”, as discussed above, we require mechanisms to combine the PNID and

FNID approaches.

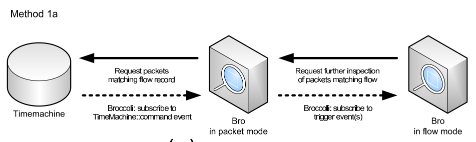
The following components were considered for forming a combined system:

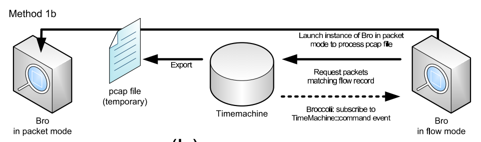
- **Software-based NetFlow probe** (nProbe): Flow information does not exist in a network a priori. It must be generated either by a NetFlow capable network device. If such a device is not available, a software implementation running on a standard PC may be used instead. As we have no influence over the networks we will be performing our evaluation on, we can not rely on the existence of such devices with an appropriate configuration. Using our own probe also ensures freely adjustable configuration parameters that may serve further comparability between the different evaluation sites.
- **Bro in flow mode**: One instance of the Bro-IDS ([29]) will use the received flow records from the probe as input and process these flows according to self-built policy scripts.
- **Bro in packet mode**: One or more instances of the Bro-IDS will run in packet mode, using the libpcap library for packet input.
- **Snort in packet mode**: Instead of the Bro-IDS running in packet mode, it would also possible to use Snort ([30]) as an alternative. However, because of the missing support for communication with the Bro in flow mode, we did not consider this in any of our combination strategies.
- **Timemachine**: The Timemachine ([31]) may serve as a temporary buffer of network traffic in order to overcome the time delay introduced by the generation and processing of the flow records. The Bro-IDS running in packet mode can retrieve the network traffic from the past for further analysis.

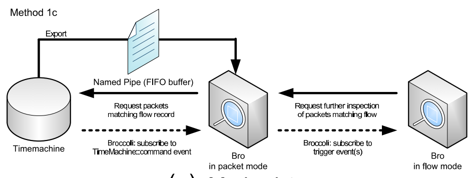
Because the Timemachine and the Bro-IDS were developed by the same group of researchers, they were designed with inter working capabilities in mind, making them ideal for use in combination. The configuration of the integrated Broccoli[4] ([32]) support, i.e. the address and port used to connect to the interfacing Bro, can be configured in the Timemachine's configuration file *tm.conf*. Queries for past network traffic can subsequently be sent to the Timemachine via Broccoli, responding with the requested packets if available. Bro already comes with a wrapper policy script (*time-machine.bro*) for communicating with the Timemachine, making its usage especially simple.

The Timemachine itself usually runs with a cutoff limit w.r.t. payload size. In order to efficiently use the Timemachine for our work with complete, raw payload on the packet level, the cutoff limit is removed. Instead, we may possibly use a similar concept in the flow part of the IDS, where only limited aggregated information is available. Another reason for the removal is the time we need access to the stored packets: While the Timemachine was designed for recording days of traffic, it is more likely that we do not need to go back more than a couple of hours maximum for our purposes because the NetFlow probe can instructed to export the flow records regularly.

[4]communication protocol between instances of Bro


Method 1a
Timemachine
Request packets matching flow record
Broccoli: subscribe to TimeMachine::command event
Bro in packet mode
Request further inspection of packets matching flow
Broccoli: subscribe to trigger event(s)
Bro in flow mode
(a) Method 1a


Method 1b
Bro in packet mode
pcap file (temporary)
Export
Timemachine
Launch instance of Bro in packet mode to process pcap file
Request packets matching flow record
Broccoli: subscribe to TimeMachine::command event
Bro in flow mode
(b) Method 1b


Method 1c
Export
Named Pipe (FIFO buffer)
Timemachine
Request packets matching flow record
Broccoli: subscribe to TimeMachine::command event
Bro in packet mode
Request further inspection of packets matching flow
Broccoli: subscribe to trigger event(s)
Bro in flow mode
(c) Method 1c

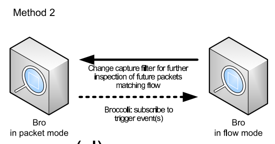

Method 2
Bro in packet mode
Change capture filter for further inspection of future packets matching flow
Broccoli: subscribe to trigger event(s)
Bro in flow mode
(d) Method 2

Figure 23: Hybrid Setups

The processing in the Bro in flow mode is designed to make queries for longer time periods unnecessary. Additionally, it is crucial to keep the query times as low as possible in order to maintain the intrusion detection close to real time. Without mentioning under what configuration, Kornexl notes in his master thesis [31] that queries take roughly 5 to 20 seconds, which is already a quite high delay for real time monitoring. The idea is therefore to run the Timemachine on a system with lots of RAM, which is very inexpensive nowadays, and to not record to the hard-drive at all. The flexible design of the Timemachine allows adapting its behavior to our needs with just a few changes in the configuration files and no changes to the code itself. It must also be noted that the Timemachine stores connections in a bidirectional manner compared to NetFlow records that are generated for every unidirectional connection. It may therefore become necessary to unify every two NetFlow records to a single, Timemachine compatible query.

Four different setups of Bro and the Timemachine were considered for experimentation:

Method 1a Method 1a (Figure 23(a)), the preferred method, uses one instance of Bro in flow mode collecting records from a NetFlow probe and one instance of Bro in packet mode, not actively capturing any packets from a network interface. The Timemachine, also running on the same machine, is configured to capture packets from a mirrored switch port at the same location in the network as the NetFlow probe. The Timemachine initiates a Broccoli connection to the Bro in packet mode. The Bro running in flow mode analyzes the received flow records using one or more policy scripts for finding suspicious flows (design of an “unsharp pre-filter”). Flow records being identified as suspicious are subject for further analysis using packet inspection. For this reason, the policy scripts define specific events that are fired in just this case. These events are subscribed by the Bro in packet mode using Broccoli, also allowing for the transmission of the flow record contents to the Bro in packet mode. If this event is fired in the Bro in flow mode, the same event is fired in the Bro in packet mode. The event handler on the flow instance is left empty (except for writing debug output to the screen or a debug-logfile). As for the instance in packet mode, the event handler specifies further actions to be taken. These are primarily forming and sending a query to the Timemachine in order to retrieve the packets that are referenced by the flow record. Depending on configuration, connection queries sent to the Timemachine can also be flagged with the subscribe property, causing the Timemachine to send all packets matching the query in the future without explicit request. The query results, in the form of network packets, are sent via Broccoli to the Bro in packet mode and are subsequently processed by one or more stock or customized policy scripts. If attacks are detected in the packets, this is logged to the screen or to files, as this is done in conventional IDS operation.

However, initial experiments revealed a drawback of this Method: If more than one type of intrusion should be monitored, which will usually be the case, the core of the Bro in packet mode will have to reidentify the traffic received from the Timemachine in order to determine which policies to apply. This may cause an unnecessary overhead, because the process of traffic identification made already by the Bro in flow mode is repeated. For this reason, Method 1b was considered as a variation to Method 1a.

Method 1b Method 1b (Figure 23(b)) is conceptually similar to Method 1a, but there is no instance of a Bro in packet mode that is running all the time. In this case, the Timemachine establishes a Broccoli connection to the Bro in flow mode. In the case of a suspicious flow, the Bro in flow mode sends a command to the Timemachine to write all packets matching the suspicious flow to a file in pcap format. The Bro in flow mode then starts an instance of Bro in packet mode that does not listen on a network interface, but rather reads and processes the contents of the pcap file.

While Method 1b holds a potential additional gain in resource efficiency, as there is no running Bro in packet mode in the case of no suspicious flows, it proved to perform poorly. Reasons for this were twofold: On one hand, the increased hard drive access, caused by the detour of the exported Timemachine query results, strained the system severely. Furthermore, it became foreseeable that the disadvantages would outweigh, as multiple instances of Bro would be launched cause an comparatively larger overhead, while periods without suspicious activities were rather scarce. Therefore, Method 1b was dropped.

On the other hand, while we experienced that the Timemachine performs well on its own in the high volume evaluation locations, substantial problems occurred once it had to process queries issued by Bro. In our high volume traffic environments, out of bounds errors occurred on the index and further queries were not processed at all until the Timemachine was restarted, once the Timemachine was configured to use more than 1209 MB of memory for buffering. The author of the Timemachine provided a patch for this problem, although further tests still showed similar problems when using a buffer size over several gigabytes.

Furthermore, when the Timemachine was confronted with several hundred queries per minute, it was unable to continue packet capturing while processing the queries, resulting in a packet drop rate of 100%. In most cases this lead to an irrecoverable deficit that remained until the Timemachine was restarted. As we were unable to determine whether this phenomenon was caused by the Broccoli communication itself or by an internal error in the Timemachine, we tested Method 1c as an alternative.

Method 1c Method 1c (Figure 23(c)) is conceptually identical to Method 1a, but uses a different communication channel for packet data between the Bro in packet mode and the Timemachine, in an attempt to reduce Broccoli communication strain on it.

In contrast to Method 1a, query results (i.e. the packets returned from the Timemachine to Bro in packet mode) are not transferred over the Broccoli connection. It is only used to send the queries to the Timemachine. A named pipe serves an alternative path for query results to the Bro in packet mode. Similar to method 1b, the Timemachine is instructed to write all of the query results to a single file which was previously configured as a FIFO buffer using the common *mkfifo* UNIX command. The Bro in packet mode reads from this buffer just as it would from a regular pcap file and processes the data accordingly.

Unfortunately, this change had no considerable effect on the packet drop issue. Therefore, a further attempt was made by disabling the buffering mechanisms of the Timemachine completely and only working with subscriptions. This implicates the assumption that we are able to detect infected hosts only on the basis of future payload, because it is likely to reappear sometime in the future. This was verified for the Storm worm and is likely to be true for other botnets. Using the Timemachine with no buffering enabled and only with subscriptions is still useful, as its querying mechanisms on specific connections are more flexible compared to the filtering options of the BPF.

As the packet dropping issue disappeared first, the configuration change showed promising results. However, the Bro in packet mode showed random segmentation faults disrupting the evaluation process. These faults could not be reproduced deterministically but appeared in combination with high query counts, sometimes only a few minutes after starting the measurement. The origin of the faults appears to be in the Broccoli communication, as the Bro in packet mode crashes with `internal error: unexpected msg type: 112`. A solution for this problem could not be found in reasonable time and was observed with the newest revisions of Bro and the Timemachine available at this time.

After having thoroughly tested these approaches, we have decided to completely eliminate the Timemachine from the combination method and used Method 2 for conducting our measurements.

Method 2 Method 2 does not use the Timemachine at all and therefore is unable to perform packet inspection on past traffic, which triggered FNID alerts. It is considered for implementation, if tests show that running the Timemachine is unable to keep up with processing queries and simultaneously capturing the network traffic or that the total delay between the reception of the flow records and sending the query to the Timemachine is so large, that the packets to be retrieved are no longer available in the Timemachine's buffer (considering buffer size in memory in the region of 4-8GB).

With method 2, it is assumed that the detection of attacks is possible by only looking packets arriving in the future. Instead of the Bro in packet mode issuing a query to the Timemachine, it captures the packets live by itself but with an initial BPF filter that excludes all traffic. If a suspicion event is fired by the Bro in flow mode, the capture filter of the packet-based Bro is reconfigured so that traffic from now on matching the connection parameters passes through the packet-based Bro and is further analyzed.

4.3.3 IRC Botnet detection

Most botnets nowadays active in the Internet still use an IRC server for command and control purposes of their bots. Storm is the big exception. For this reason a method had to be found that could easily be implemented for establishing an adequate suspicion on the basis of flow records. While actual IRC traffic of chatting users normally runs on public IRC servers that run on port 6667/tcp or 6668/tcp, this must not necessarily be the case for botnets. Therefore the challenge was to find a method for identifying IRC traffic on arbitrary port not having any payload data available. Because botnets may use IRC servers hosted any arbitrary ports, we are not able to define a restrictive *precondition* w.r.t. ports. As IRC is a TCP based protocol, we can at least restrict our analysis to flow records of TCP connections. For the Bro in packet mode, we prepend the BPF filter with the `tcp` filter element.

NetFlow Pre-Filter As described in [33], IRC connections exhibit so called IRC ping-pong messages. The IRC server sends a ping message containing the server hostname to the client. The client then responds with a pong message, also including the server hostname. The reason for these packets is to check whether the client is still alive; it is used as a keep-alive function that is also popular amongst other protocols. As our current intention in this section is not the flow based detection, but to gain an initial grasp on possible resource gain or loss w.r.t. scalability of hybrid concepts, we will not implement the whole framework of [33], but to start by implementing a IRC ping-pong detector for Bro using only flow data, in order to locate idling IRC connections.

The reason why ping-pong traffic can be detected with only flows is that it expresses a constant amount of packets and octets for each time such a message is exchanged. [33] defines this as the bytes-per-packet ratio which stays constant in this case. Bots connected to the IRC server are often idle, awaiting commands from their master, i.e. there is no other communication with the IRC server taking place except for the ping-pong traffic. When a bot is active, this method of detection may cease to function.

The primary state model used for analysis of these periodic flow records is a hierarchical table which we call **observ**ation table. The advantage of the hierarchy is that lookup operations on the tables are naturally inexpensive, which was verified in test runs in the CSG Testbed. The elements of the *observ* table are set to expire 30 minutes after their last update, so that memory consumption can be kept at an acceptable level.

The *oberv* table construct is shown in figure 24: The outermost table is indexed by the source host, carrying tables indexed by source port that in turn carry tables indexed by the destination host. Next in hierarchy are tables indexed by destination ports, flow *pkts* value and flow *octets* value. The innermost table carries an occurrence counter that is incremented for every retrieved flow record with equivalent values.

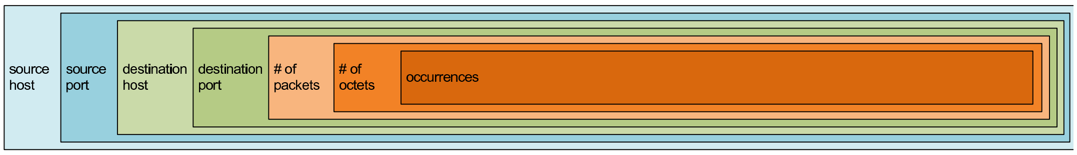


Figure 24: The flow observation table

The following imaginary *observ* table example shows that a flow from host 10.25.94.31, TCP port 1190 to host 192.168.61.39, TCP port 16149 consisting of 1 packet with a size of 46 bytes was reported 3 times:

```
[10.25.94.31] = {
   [1190/tcp] = {
      [192.168.61.39] = {
         [16149/tcp] = {
            [34] = {
               [19838] = 1},
            [1] = {
               [46] = 3 },
            [11] = {
               [920] = 1 }
            }
         }
      }
   }
```

The *idxObserv* method is called for each new arriving flow record from the *netflow_v5_record* event. We do not make use of the flow restitching function (i.e. we do not use the *new_flow* as in the peer-to-peer case), as we are interested in the packet and octet values that may change when flows are updated.

The *idxObserv* function is organized as follows: The flow record is first checked for a number of preconditions. IRC traffic is only found on TCP ports, which is why flow records of non-TCP communication are discarded. Because idling connections only transfer a small number of packets and octets, all flows with more than 6 packets and/or 200 bytes are also discarded. These values were determined empirically by manually analyzing IRC ping-pong messages. These settings include more traffic than necessary, but their function as a precondition is to avoid blocking of processing by large volume flows as early as possible. The *observ* table is then hierarchically searched top-down. If elements do not exist yet, they are created and filled with the data from the current flow record. The occurrence counter at the end is set to 1 or increased by 1 if all parent elements already exist. After updating the *observ* table, the elements are fed into one or more classifiers in order to determine if the flow is likely to carry IRC traffic.

The first classifier developed was designed with the idea in mind that the ping and the corresponding pong message have their source and destination addresses and ports reversed, that for each message one packet is used and that these are similar in size. The range was set to plus or minus 5 compared to the corresponding message. During test runs it turned out that the first assumption is true, but that the number of packets actually differs. The first classifier was therefore discarded.

The second classifier also first checks flows with reversed source and destination addresses and ports compared to the newly received flow. As manual inspection showed, the ping message usually consists of one message but the ponging host sends an additional “empty” ACK packet. Therefore the classifier checks next for flows that have an amount of packets larger or smaller one compared to the current flow. If such flows exist, next flows are checked that have the same number octets with a tolerance of plus/minus one byte. If such a flow exists, the classifier finishes and marks the flow as suspicious to be an idling IRC connection, thus the *broccoli_analyzethis_periodic* event is fired. This event is constructed identically to the *broccoli_analyzethis_flood* event in the peer-to-peer use case.

Tests in the CSG Testbed with open connections to public IRC servers were able to detect all IRC connections with a number of false positives. Interestingly, keep-alive behavior of other protocols was detected as well. In one case, a unidentified custom application running on PlanetLab, whose packets carried the payload “keepalive”, were marked suspicious. The other case were idle SSH connections, i.e. SSH sessions with server that were open but not currently used by the user. The amount of false positives was small enough to make this an efficient pre-filter. Resource usage was at an acceptable level.

Signature matching We do not focus on a specific type of botnet because they are numerous active botnets without one being particularly prominent at this time.

For the Bro in packet mode, the Broccoli wrapper and BPF manipulation mechanisms from the peer-to-peer use case are used as a starting point. Furthermore, the *irc.bro* and *irc-bot.bro* policy scripts shipping with Bro are used. In order for Bro to consider inspecting traffic on arbitrary ports with its IRC analyzers at all, Bro’s Dynamic Protocol Detection (DPD) functionality is activated by loading the *dpd.bro* script.

The *irc.bro* script writes all IRC traffic in human-readable format to a log file, allowing for manual inspection of bot presence. The *irc-bot.bro* script does so in a separate file but only logs messages of some well known bot behavior, identified by message content or special nicknames. No information is available about what bots exactly are detectable by this script.

In addition to the policy scripts, we converted several Snort rules for detecting IRC bots from Bleeding Edge Threats. Some of these rules rely on well known nicknames and bot commands, similar to the *irc-bot.bro* script, while containing unique signatures for specific bots. Because of the *snort2bro* conversion limitations, we were not able to convert all rules for use with the Bro in packet mode. In the end, we had 68 rules available with specific rules for *Agobot/SDBot*, *pBot (PHP bot)*, *perlb0t/w0rmb0t*, *B0tN3t*, *Kaiten*, *Pitbull*, *W32.Virut.A*, *iroffer*, *SpyBot* and *GTBot*.

4.3.4 Peer-to-Peer Botnet detection

Because we are able to evaluate their detection confidence in real-world sites, the evolving threat from peer-to-peer based botnets makes them an ideal scenario for a combined NetFlow and packet-based IDS approach. Peer-to-peer traffic, whether malicious or not, is found on arbitrary ports, making static filtering attempts based on ports useless (such as BPF). In this work, we define the most popular use of peer-to-peer, file sharing, as benign traffic, though this differs from the definition in many company policies. If the primary focus for such a system is laid on a commercial end-user ISP, peer-to-peer traffic accounts for a large part for the transferred volume. The Storm (aka Peacomm) worm was chosen as a specific scenario, as is believed to be the most active botnet at this time, guesses varying between several ten-thousand to several million infected hosts in the Internet [34]. Storm, in its current state (January 2008) uses an encrypted version of the eDonkey/Overnet peer-to-peer protocol for command and control communication. Because some parts of this encrypted communication contain static values for which signatures exist, this does not affect its eligibility as a scenario.

General research on the Storm worm is not common to find, although there is one paper that has an extensive look at the internals of the worm, focusing on its program code but also on network communications [35]. There are also several Snort rules published in the paper that should help detecting the worm. Extensive research on the worm's internals, specifically on its rootkit behavior, has been conducted by Frank Boldewin [36]. In addition, some Snort rules were found at Bleeding Edge Threats [37], a company that focuses on delivering signatures for current attacks.

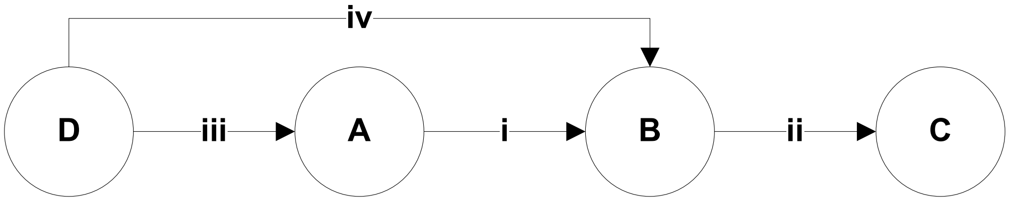
When speaking of communication patterns of botnets, we have to distinguish propagation from command and control traffic (cf. background section of [38]). The Storm worm has no automated way of propagation, but misleads computer users to manually download and execute itself by sending SPAM e-mails containing download links. Some of the participating bots are actually involved in sending these e-mails while others act as web servers where the infected executables are downloaded from. Sending e-mails (i.e. outbound traffic on TCP port 25, SMTP) and hosting a web server (i.e. inbound connections on TCP port 80, HTTP) is not unique to the Storm worm and is in fact often benign. For example, many users choose to host their web and mail servers at home on their DSL connection. Implementing a NetFlow-based pre-filter on these propagation characteristics would therefore results in a very high number of false positives. Additionally, the contents of the propagation e-mails changes regularly and no adequate payload signatures exist for detecting Storm propagation with an acceptable alert confidence.

We therefore exclusively focus on the detection of Storm's peer-to-peer based command and control communications. Suitable signatures for packet inspection are available, but no directly implementable NetFlow-based pre-filters exist.

In order to achieve comparable results w.r.t. resource consumption, traffic will have to be reduced to a level that may be handled by a standalone Bro in packet mode as well. Thus, while peer-to-peer traffic by itself is not bound to specific ports, we are considering the fact that most peer-to-peer systems such as BitTorrent and eDonkey, as well as the Storm worm use UDP connections with source and destination in the port range between 1024 and 65535. Therefore, flows not matching this criteria are discarded before any other processing. We refer to this simple form of a lightweight first filter as the *precondition*. For all flows describing connections matching the precondition, further processing is done. The precondition is set as a BPF filter for nProbe, the Bro in packet-only mode and is prefixed for the Bro in packet mode. Because Storm may alter its communication ports, this is potentially dangerous. However, we can not perform adequate measurements without it. Waiving the precondition would cause nProbe and the Bro in packet-only mode to exhaust the system's capacity, also affecting the Bro in flow and packet mode, which will possibly falsify the evaluation results.

We will consider four approaches for P2P traffic detection:

Diameter This approach is based on a method developed in [39]. In this paper, the evaluation was performed not with flow, but with packet data. Interestingly though, the approach uses data that is completely available from flow records which makes it an ideal candidate for implementation in this use case. Also, evaluation results show very good detection rates of peer-to-peer traffic, with false positives in the region of only a few percent. The key concept of this approach uses two classifiers for identifying peer-to-peer traffic: 1. Large diameter, 2. Many nodes active as both, client and server simultaneously. While the second classifier is straight-forward to understand and implement, the diameter classifier may need further explanation.



host	level
A	0
B	1
C	2
D	-1

estimated diameter: 3

Figure 25: Estimation of the Network Diameter [39]

The diameter is approximated by measuring each host's level, which describes the time of its first record. Hosts initiating connections before accepting any are given the level 0. Hosts that accept connections are given one level higher than their corresponding initiators. Hosts that initiate connections to hosts that already have a level assigned are given one level lower than their corresponding accepting hosts. Once a host is assigned a level, it is not changed anymore. The calculation of the diameter is made by subtracting the maximum from the minimum level. The authors state that the employed method is only an approximation of the true diameter, but sufficient for these purposes. An example of a network diameter estimation is shown in figure 25.

The threshold settings of the number of hosts acting as clients and servers and the diameter value may be adjusted to match the network environment. The evaluation results show that a diameter threshold of 3 and a client/server threshold of 3 is reasonable to begin with. Calculation of these values is always done on the basis of a specific server port (i.e. the destination port of a flow). This is somewhat problematic, as the output of the classification are not flows but rather just ports identified to likely carry peer-to-peer traffic. This is why we had to build an additional wrapper to cache the received flows (*span* table) to retrieve the flows describing communication on this port.

The implementation of the described algorithm was a rather difficult task to realize in Bro's policy scripting language, because of the constraints already mentioned earlier. The first attempt was to transform the pseudocode (printed in [39], Figure 4) to Bro. The attempt failed, as the pseudocode is explained incompletely in the text. The second attempt was therefore to directly adapt the graph model introduced in the paper: For every server port P a separate graph is maintained, consisting of vertices V and edges E. Vertices represent the hosts using port P for communication, while edges represent connections between hosts. Thus they are stored in the tables *GpV* (index: server port P, element: set of addresses) and *GpE* (index: server port, element: set of address pairs) respectively. The level of a vertex/host is stored in the *GpVlev* table (index: port, element: table indexed by addresses containing levels (ints)). Additionally, for every vertex, the incoming and outgoing neighbor hosts are maintained. These are stored in the *GpVin* and *GpVout* (both, index: server port P, element: table indexed by addresses containing sets of addresses) tables. We strongly relied on a hierarchical model of tables and sets in order to keep the data well structured for later processing.

Further, some supportive variables are introduced: *GpMinLevel* and *GpMaxLevel* (index: server port, element: level (int)) contain the current minimum and maximum level values that are used for the calculation of the diameter on that port. *GpLevCount* (index: server port, element: table index by int containing count) is used to count the number of times a certain level occurs on a specific port P. Finally, *reportedPorts* is used in conjunction with an expiry time to avoid duplicate reports of ports.

The *new_flow* function calls the *addConnection* function which contains the code to create and update the server port graphs according to [39]. Further, the *assignLevel* function assigns levels to the hosts. The supportive functions *updateMinMax* and *updateLevCount* called from the *addConnection* function are responsible for updating the *GpMinLevel*/*GpMaxLevel* and *GpLevCount* values, respectively. Before the *addConnection* function is completed, it checks whether the graph exceeds the server/client and diameter thresholds. If so, the *span* table is checked for connections from and to port P and *broccoli_analyzethis_flood* events are fired for each host involved in communication on this port.

Test runs in the CSG testbed showed that the algorithm indeed detects peer-to-peer communication, but the operations for maintaining the rather large amount of tables is very expensive. CPU load rises to 100% within short time after launching. Memory consumption is significant and caused several crashes. The script was therefore optimized, as it contains many redundant data. As a result, the *GpV*, *GpE* and *GpVout* tables were completely commented out. The elements of the *GpVin* and *GpVlev* were set to expire 10 minutes after they were created. Omitting an expiry timeout would result in constant growth of the tables and therefore in gradual starvation of resources. The other tables do not have a expiry timeout set, as their maximum number of elements is the amount of ports matching the precondition, i.e. 65535 - 1024 = 64511.

Static threshold, sliding window The usage of a static threshold was chosen to be implemented first, as it is the most simple approach. The threshold (*span_limit*) is stored in a global constant of type *count* and is set manually beforehand by the operator.

Bro provides flexible functions for letting table elements expire, depending on the time passed since the creation of the entry, the time passed since last read access or the time passed since last write access. Keeping track of the number of connections matching the precondition by creating a table of type *count* indexed by source address is possible, but does not offer the possibility of a "sliding window" with the provided expiration functions, meaning we are unable to count the number of connections during the last, for example, 10 minutes. We therefore developed a workaround for this problem by creating a *span* table[5] of booleans, indexed by *conn_id*s. The entries in the *span* table are set to expire

[5]The boolean type only serves as a placeholder function, as a table can not be left empty. Bro features

after a specified amount of time they were created (i.e. they are deleted from the table). Together with the threshold value, the expire value needs careful adjustment.

In the *new_flow* function we first check the precondition. If the flow passes, we write the *conn_id* of the flow to the *span* table (and set the boolean value to true). We then call the *test_span* function in order to check if the threshold for the local-network host is exceeded.

In order to save resources, the *test_span* function first checks if the host is present in the *span_marked* table and then loops over the *span* table to count the number of *conn_id* elements having the same local-network host as source (*orig_h*) or destination (*resp_h*) value. If this value exceeds the threshold, the *broccoli_analyzethis_flood* event is fired and the host is added to the *span_marked* table.

In initial test runs in the low-volume CSG Testbed, we observed that looping over the *span* table every time a new flow arrives is very expensive and CPU utilization reaches 100% shortly after starting Bro. It was therefore decided that a different method must be used in order to cope with the even higher number of arriving flows to be expected in the high-volume environments.

Static threshold, fixed window This approach is essentially based on the same idea, but the usage of a “sliding window” was discarded because of the experienced performance issues. The *span* table is now indexed by the local-network host IP and carries elements of type *count*. The *count* value specifies the number of flows originating from this host that match the precondition. It is increased upon arrival of new, matching flows. The elements of *span* are set to expire *n* seconds after they were created. This concept reflects the idea of a “fixed window”: Flows are counted during *n* seconds. If the threshold is exceeded during this time, it is eligible for further analysis, if not, the *count* value is reset to 0 and a new window begins.

Again, the starting point is the *new_flow* function. First of all, the precondition is checked. If the flow passes it, the *span* table is checked for an already existing element for the local-host host IP address (i.e. the *orig_h*/*resp_h* from the *conn_id*). If this is the case, the counter value is increased by one. If the host is not present yet, it is added to the able as a new element with a counter value of 1. The table element’s *count* value is then checked whether it exceeds the threshold value. If this is the case, and the host IP is not yet present in the *span_marked* table, the *broccoli_analyzethis_flood* event is fired and the host IP address is written to the *span_marked* table.

Test runs in the CSG Testbed have shown a significant decrease in CPU utilization, which is in the order of a few percent. The reported hosts that were eligible for analysis with the Bro in packet mode were largely the same as in the “sliding window” case. It is therefore assumed to scale better for use in a high-volume network environment.

The fixed window poses a potential issue because bursts of increased flows may not get detected when they fall between two windows. In this case, the threshold will not be exceeded and the host will not be marked suspicious. Because Storm, once it has started its command and control communication, continuously opens new connections, it is unlikely to get missed because of this issue.

the *set* type exactly for this case, but we have to use an “empty” table instead because it lacks the expiration functions of tables.

Adaptive threshold Static thresholds are easy to implement, but they do not adapt to different networks and are therefore not universally usable without manual adaption to the environment. While it can be seen as acceptable for an IDS operator to adjust this one parameter a single time, a policy script adapting itself automatically to the environment is more desirable. As a common statistic dimension, we have decided to use the average[6] value of the number of flow matching the precondition as a basis to adaptively calculate the threshold.

This average value is then multiplied with the *overavg* factor. The *overavg* factor was initially set to 10. This sets the threshold to the average value of flows per host matching the precondition multiplied by 10. In the case of the CSG testbed, this resulted in an threshold value between 40 and 100, depending on weekday and time. Hosts infected with the Storm worm display values of at least a few hundred precondition matching flows. Therefore, the *overavg* factor needs careful adjustment, just as setting the threshold directly. We suspect that depending on the characteristics of the infected system and its Internet connectivity parameters, determining a universal valid fixed threshold value for Storm infected hosts in all networks is not possible. Therefore, the *overavg* setting may be more universal than the threshold itself, releasing the operator to manually determine an appropriate threshold value by making test runs. However, in this case we make the assumption that the amount of hosts involved in peer-to-peer activity is similar in all networks. If a large part of the hosts connected to the network are infected with Storm, the adaptive threshold may rise to a value that will make the Bro in packet mode miss most infected hosts.

Most of the design has not changed for the adaptive threshold script. Again, the *new_flow* function, the *span* table and the *span_marked* table are used in the same configuration as with a static threshold. The threshold value is initially set to 1000 so that there is no event burst when the script is starting and no average value can be calculated yet. Because the calculation of the average value can be a resource intense task, it is not launched every time the *new_flow* function is called but set to run every 60 seconds, except for the first time when it is run 30 seconds after starting Bro. This *update_threshold* function essentially loops over the *span* table to calculate the total number of flows. The total number of flows can be retrieved by using Bro's *length* function on the *span* table. The average value is then calculated as the total number of flows divided by the total number of hosts. The new threshold value equals the new average value multiplied by the *overavg* value.

If a flow is eligible for further analysis with the Bro in packet mode by firing the *broccoli_analyzethis_flood* event is determined with the same code described in the static threshold case.

Test runs in the CSG Testbed have shown that resource consumption is comparable with the static threshold case, although short CPU peaks can be observed during the periodic threshold value calculation.

Top n% The pragmatic idea behind this approach is reducing the number of packets that need to be analyzed by only considering the flows originating from hosts that are responsible for the n% most flows. Like with an adaptive threshold, we assume that this

[6]We refrained from considering the median value because its calculation requires sorting the *span* table, which is not efficiently implementable in Bro's policy script language.

approach may be more universally deployable in different network types under the assumption that the amount of hosts infected with Storm is not excessive. As Bro's policy language is somewhat constrained, it is not straight forward to sort tables according to their values. A concrete example is the *for* command that can only be used to loop over table elements, but not in a particular order. Sorting had therefore to be implemented in a recursive fashion.

As sorting the *span* table, which is used in the same configuration as in the previous approach, is an expensive operation, it is conducted only in discrete time intervals. The *getmaxtenspan* function is called in 60 seconds intervals. All elements from the *span* table are then copied to the *span_copy* table, the *maxcounter* variable (type *count*) is set to zero and the *tenper* variable is set to the number of flows in *span* multiplied with the *topn* factor to get the absolute number of flows to retrieve. The *getmaxspan* function is then initially called. It loops recursively over the *span_copy* table, each time firing the *broccoli_analyzethis_flood* event for the largest element in *span_copy* and then deleting it from the table. This way, the next time the function calls itself, the seconds largest value will be processed correspondingly and so on. Each time the function is called, a counter is incremented. If this counter exceeds the *maxcounter* value, the sorting terminates. As in the previous approaches, each processed flow is kept in the *span_marked* table in order to avoid duplicate firing.

Test runs in the CSG testbed have shown resource consumption comparable to the adaptive threshold case.

Signature matching Signatures from two different sources were initially considered to be used in our experiments:

- SRI signatures
- Bleeding Edge signatures

The **S**RI signatures of [35] describe the network dialog interaction of Storm. For three of these dialogs Snort signatures are provided:

- **Outbound attack propagation events**: Describes the communication of local bots in attempts to infect other systems. In the case of Storm this is SMTP communication (from a not-SMTP-server local asset) to numerous external SMTP servers. The e-mail messages transferred in this SMTP communication contain links to servers running on other bots that shall trick users to download infected executables.
- **Local asset attack preparation events**: Describes the communication of bots occurring in preparation of attacking other local assets. In the case of Storm this may be an increased number of DNS MX queries to the local DNS server, i.e. lookup for SMTP server addresses.
- **Peer coordination events**: Describes the command and control communication in a botnet, containing instructions directed to the bots what activity they should perform. As Storm is peer-to-peer botnet, it does not receive these commands from a central entity but from several other peers within the larger botnet. Storm thereby leverages the eDonkey/Overnet protocol for peer coordination.

The first two dialogs are universal to Internet e-mail communication, their intrusion detection rules are based on the occurrence of SMTP communication between at least one host not explicitly defined as an SMTP server in the network. This is very popular amongst Snort rules, but its use in backbones or ISP networks is rather difficult. Customers may choose to run legitimate SMTP servers on their lines for their e-mail communications. Therefore the rules for outbound attack propagation and local asset attack propagation were not considered for implementation.

The peer coordination event rule though, uses a unique regular expression defining the peer-to-peer traffic that is unique to Storm. The signature comes in two rules, one for the Overnet file search command:

```
alert udp $HOME_NET 1024:65535 -> $EXTERNAL_NET 1024:65535
(msg:"E7[rb] BOTHUNTER Storm(Peacomm) Peer Coordination Event
[SEARCH RESULT]"; content:"|E311|"; depth:5; rawbytes;
pcre:"/[0-9]+\.mpg\;size\=[0-9]+/x"; rawbytes;
classtype:bad-unknown; sid:9910013; rev:99;)
```

and one for the Overnet file publish command:

```
alert udp $HOME_NET 1024:65535 -> $EXTERNAL_NET 1024:65535
(msg:"E7[rb] BOTHUNTER Storm Worm Peer Coordination Event
[PUBLISH]"; content:"|E313|"; depth:5; rawbytes;
pcre:"/[0-9]+\.mpg\;size\=[0-9]+/x"; rawbytes;
classtype:bad-unknown; sid:9910011; rev:99;)
```

The rules were converted for use with Bro using the *snort2bro* script. The content bytes were converted automatically by the script, the regular expression (pcre) had to be adapted by dropping the concluding */x* parameter which signifies that whitespace characters shall be ignored. It is unclear whether this has any effect on the rule's accuracy.

Tests with an infected virtual machine showed that the peer coordination rules from SRI were not able to detect any Storm infected hosts at all. Therefore, the traffic was manually inspected. None of the regular expression parts the rules rely on were actually found in the traffic. It is therefore suspected that Storm communication has changed since the publication of these rules and that they are outdated. It is indeed known that Storm communication was clear-text at first and changed to encrypting the communication later [40]. We therefore retrieved adapted signatures for the encrypted traffic from a different source.

Bleeding Edge Threats signatures of the Bleeding Edge Threats [37] community, which provides Snort signatures for newly appearing security threats. The repository contains many signatures, also for the Storm worm. 10 signatures were isolated and converted for use with Bro and then tested against the infected virtual machine. As it turned out, only two of these signatures were able to detect the infected host:

```
alert udp $EXTERNAL_NET 1024:65535 -> $HOME_NET 1024:65535
(msg:"BLEEDING-EDGE TROJAN Storm Worm Encrypted Traffic Inbound -
Likely Search by md5"; dsize:25; threshold: type threshold, count 40,
seconds 60, track by_dst; classtype:trojan-activity; sid:2007636; rev:1;)

alert udp $HOME_NET 1024:65535 -> $EXTERNAL_NET 1024:65535
(msg:"BLEEDING-EDGE TROJAN Storm Worm Encrypted Traffic Outbound -
Likely Connect Ack"; dsize:2; threshold: type threshold, count 10,
seconds 60, track by_src; classtype:trojan-activity; sid:2007637; rev:1;)
```

These rules are somewhat problematic, as they do not contain any payload signatures. They rely exclusively on the repeated appearance of a specific payload size in packets (*dsite* parameter) originating from the same source (*track by_dst*) during a specific time frame (*threshold* parameter). The author of these Snort rules annotates them with the comment that they cause a large number of false positives by hosts running Skype. An additional problem is that *snort2bro* is unable to convert Snort's *threshold* function for use with Bro. However, packet inspection is still necessary for the functioning of these rules. With flow data only, we do not get any information of the payload size per individual packet.

In contrast, Bro's signature processing framework (*signatures.bro*) provides a similar feature to Snort's *threshold* function. By applying the *SIG_COUNT_PER_RESP* setting on those rules, we can count the number of times a specific source host triggers them in packets directed to individual destination hosts. The infected virtual machine triggered "Search by md5" rule on over 1000 destination hosts in a monitoring window of 5 minutes, while the "Connect Ack" rule was triggered for over 100 destination hosts. A cross check with a host running Skype revealed that the rule is indeed triggered, but for no more than 10 destination hosts during normal usage. Hence, with these empirically determined thresholds, we are able to reliably detect hosts infected by the Storm worm. In the evaluation, we count alerts of hosts infected with Storm only if both rules were triggered for at least 50 hosts.

4.3.5 Summary and Outlook

In this work, we are adressing the question, whether the advantages of the data reduction of Flow-based Intrusion Detection may be combined with the advatages of the higher confidence degree of Packet-based Intrusion Detection, in order to increase the resource efficiency on the search for security relevant intrusions.

In order to follow this question, we set up installations on two measurement points, characterized them for comparability and evaluated mechanisms for hybrid approaches, as well as suited detection approaches for both IRC-based and P2P-based botnets.

While this work is still in an early stage, we will continue by evaluating both resource consumption as well as detection accuracy of the hybrid approach in comparison to a traditional purely packet-based approach. To this end, we envision running both the hybrid approach, as well as the traditional approach simultaneously on live traffic, in order to achieve direct comparability.

4.4 Annotated bibliograph

An annotated bibliography with papers about large-scale, flow-based and sample-based intrusion detection has been created. The bibliography currently comprises 21 entries and is available in the SVN repository at:

`https://svn.man.poznan.pl/svnroot/emanics/wp7/snid/snid_papers.bib`

5 Scaling of Network Peer Services Employing Voluntary Cooperation (SNPS)

This section addresses the issue of scalability for network peer services employing voluntary cooperation. We explore the meaning of scalability in terms of fundamental relationship between workflow, processing and uncertainty, in the context of autonomous nodes/servers interacting and providing services among themselves on a voluntary basis. This follows up preliminary work in refs. [41, 42] and is inspired by work on Network Patterns and on the Generic Aggregation Protocol (GAP) described in refs. [43, 44, 45]. Network Patterns are tree-like overlay networks (usually trees) designed to collate information from a topologically constrained network, such as a fixed routing infra-structure or ad hoc substrate. The structure of such a network could be constrained by geography, design, access or even by wireless power limitations. Patterns are the basic structures used in routing and switching, like spanning trees, and they are used for both dissemination and aggregation of data, from a complete connected network to a single point. Autonomic computing [46] has become a major paradigm for dealing with the growing complexity of systems and networks. It advocates greater decentralization of autonomy and only weak coupling of nodes/servers through cooperative communication. We analyze in this section the scalability of Network Patterns in the context of autonomous nodes/servers. We model Network Patterns using the framework of promise theory, where all the nodes are considered as fully independent i.e. they cannot be forced into behaviour externally. We then analyze the properties of the resulting promise structure: in particular, the sampling process and the propagation of data from node to node. We finally evaluate the performances of the star pattern (maximum width network pattern) through an extensive set of experiments, in the context of a load balancing architecture.

5.1 Scalability and Propagation

What do we mean by scaling then? We shall identify *scalability* with the ability of a given structure to deliver a consistent level of service as the number of participants in the system N grows. Already this is ambiguous: what do we mean by the service, and what is a consistent level? Are we willing to sacrifice the quality of the result for a quick response?

Why would a system "scale" or not scale? The answer to this question lies in the constraints that are imposed on the work within the network. Phrased differently, we should ask: what promises have to be kept in order to say that the system is working sufficiently well, and can we keep these promises as N becomes large?

We shall see below that the important limitation of a network lies in their distributed nature and the fact that information takes a finite amount of time to propagate. There are many decisions to be made and many uncertainties at each node, all of which make the predictability of workflow less than obvious. Nevertheless, we shall see that the basic problem of workflow never really exceeds the simple considerations discussed in ref. [41].

In distributed systems there are many factors that limit performance. Processing rate, latency and predictability are the most obvious amongst these. These factors are interrelated when the network is working together in a cooperative fashion, as dependencies

Originally navigation patterns were designed to collect data from a network of routers, whose physical topology was fixed, by creating an adaptive spanning tree to collect values in a rational manner [43, 44, 45]. It is clear however that the technique has a wider applicability especially in forming logical cooperative patterns in ad hoc (and) sensor networks, where the problems of topology constraint imply further costs and therefore the limitations of the relaying structure are more prononced.

For concreteness, we shall consider the issue of processing rate versus the uncertainty inherent in a data aggregation process. We frame the discussion in a regime of regular updates as is typical in monitoring, in order to show the strengths and weaknesses of different policy choices. In particular we study the effect of individual behaviour (modelled as voluntary cooperation) between nodes in a network on the reliability of the computed aggregates.

5.2 Promise-Based Modelling

Network patterns are designed to work in an obligation regime, where requests are pushed from one or more nodes through the network in order to aggregate data at a single end point. The "push" nature of the requests implies an automatic synchronization between sender and receiver and the issue of clock synchronization is suppressed.

However, we want to show that the issue of time relativity is central to the problem of distributed monitoring. This motivates us to look at distributed cooperation from the viewpoint of voluntary cooperation ("pull" approaches) for two reasons: first, this allows us to discuss access controls and latencies in a rational manner, and second it generalizes the patterns to encompass regular maintainance regimes like cfengine, where data are measured with a kind of heart-beat, taking the pulse of the system on a regular basis.

Promise theory is a high level description of constrained behaviour in which ensembles of agents document the behaviours they promise to exhibit. Agents in promise theory are truly autonomous, i.e. they decide their own behaviour, cannot be forced into behaviour externally but can voluntarily cooperate with one another[47]. A promise is a directed edge $a_1 \xrightarrow{b} a_2$ that consists of a promiser a_1 (sender), a promisee a_2 (recipient) and a promise body b, which describes the nature of the promise. Promises made by agents fall into two basic categories, promises to provide something or offer a behaviour b (written $a_1 \xrightarrow{+b} a_2$), and promises to accept something or make use of another's promise of behaviour b (written $a_2 \xrightarrow{-b} a_1$). A successful transfer of the promised exchange involves both of these promises, as an agent can freely decline to be informed of the other's behaviour or receive the service.

Promises can be made about any subject that relates to the behaviour of the promising agent, but agents cannot make promises about each others' behaviours. The subject of a promise is represented by the promise body b. Finally, the *value* of a promise to any agent is a numerical function of the constraint e.g. $v_{a_1}(a_1 \xrightarrow{+b} a_2)$, and is determined and measured in a currency that is private to that agent. Any agent can form a valuation of any promise that is knows about.

The essential assumption of promise theory is that all nodes are independent agents, with only private knowledge (e.g. of time). No node can be forced to promise anything or behave in any way by an outside agent. Moreover, there are no common standards of knowledge (such as knowing the time of day) without explicit promises being made to yield this information from a source. This viewpoint fits nicely with our view of collection of distributed information for measurement purposes.

We shall consider the following promise designations: $+d$ Server provides data, $-d$ Client receives/uses data, $+a$ Branch node aggregates data, $+t$ Server provides time/clock, and $-t$ Client uses time/clock. Although we speak mainly of network nodes below, it will be understood that each node is modelled as an "agent" in promise theory parlance.

We assume an underlying network substrate with partially reliable communication. The interactions between the agents form directed graphs which can be typed with the promise labels. A forward pointing promise graph forms a transpose adjacency matrix

$$a_i \xrightarrow{b} a_j \quad \Leftrightarrow \quad A_{ij}^{\mathrm{T}\,(b)}. \tag{5}$$

Since the communication requires bilateral $\pm b$ promises, there is an implicit promise graph of opposite sign whose structure is the transpose of the matrix above. Nodes can, in principle, have any in- or out-degree greater than zero. It is conventional to restrict network patterns to tree structures however so that the data flow promises of type $+d$ form a monotone confluence. The adjacency matrix $A_{ij}^{(-d)}$ for the forward part of fig. 26 and similar ones can be used to represent the typed node degrees through the relations:

$$\begin{aligned} A_{ij}^{+d} &= A_{ij}^{(a)} = \left(A_{ij}^{(-d)}\right)^{\mathrm{T}} & (6)\\ k_i^{(+)} &= \sum_j \left(A_{ij}^{(+)}\right)^{\mathrm{T}} = \sum_j \left(A_{ij}^{(-)}\right). & (7)\end{aligned}$$

Figure 26 shows the overlain complementary trees of $\pm$ data promises.

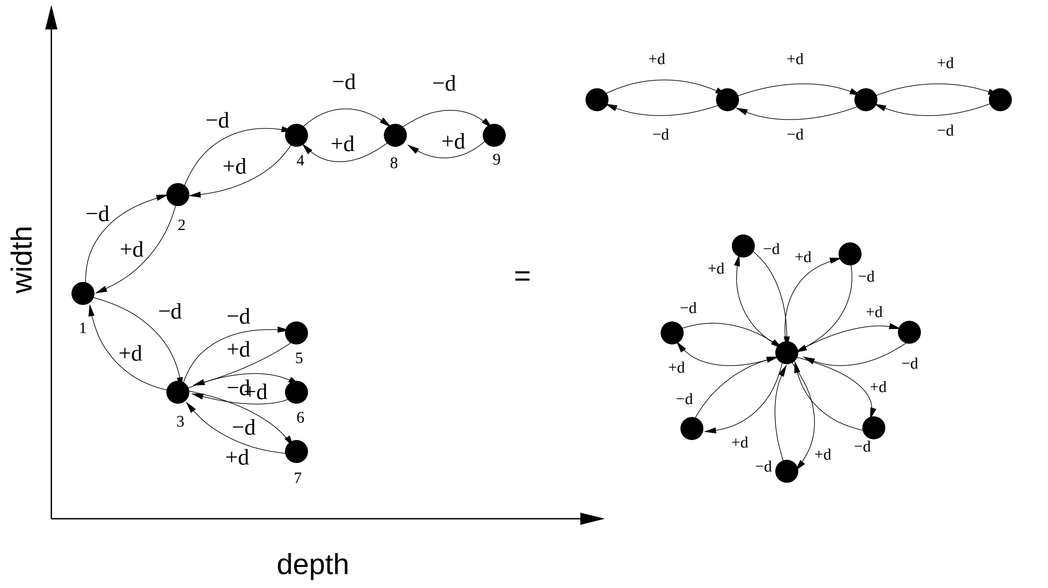


Figure 26: A bilateral promise tree of type $\pm d$ indicating "depth" and "width". The structure can be thought of a cross between the star topology and a chain, which are the extreme cases.

5.3 Kinematical Results

The rate and timing of the information flowing through the promise network structure is important to its observable properties. There are two processes taking place that ‘interfere’ with one another.

- Sampling process at each node.
- Propagation of data from node to node.

If the stream is not synchronized by waiting, information will not propagate faithfully along the full length of the chain. This is obvious in a one-shot enactment of the promises, but the matter becomes much more interesting in the case where the promises are implemented in a periodic schedule. If nodes have memory of previously measured values then one can obtain the illusion of propagation of data, but at the price of having undetermined or inaccurate results.

We begin by considering the basic rates of transport in the promise structures. We must consider the time it takes to transport across network depth and the time it takes to aggregate across network width as separate processes. By dimensional analysis[48] we know that the behaviour of the system will depend essentially on the various dimensionless ratios that can be formed from the basic scales of the system.

Let q_i be a variable at node a_i and $\frac{dq_i}{dt}$ be the rate at which q_i changes in time. We shall assume that time increases at the same rate for all observers, within the limits of measurement; in other words, all agents have clocks that run at the same rate. This is a more likely approximation to the truth than assuming that they are all synchronized. Finally, we let Q_i mean the size in bytes of the representation of the value of q_i that has been measured. This is important to know when transporting the data over fixed capacity channels.

We shall use t_i to mean the time at which a_i sampled the value of q_i, and we shall let P_i be the periodic sampling rate of the agent node a_i, so that after ℓ samples, the time will be $t_i + \ell/P_i$.

This generality is suggestive to later generalization of the results in this work. For the present, however, we shall consider the idealized case in which all agents sample at the same basic rate P, the period of the system.

Promise $a_i \xrightarrow{+d} a_j$

Let $C_{ij}^{(+d)}$ be the rate at which a_i can transmit data to a_j. This is a property of the communication channel between the agents as well as the agents’ own limitations. We now introduce the promise-valuation function $t(a_i \xrightarrow{b} a_j)$ to be the time to complete the promise $\langle a_i, b, a_j \rangle$. Then the time to transmit the data is:

$$t\left(a_i \xrightarrow{+d} a_j\right) = \frac{Q_i}{C_{ij}^{(+d)}}. \quad (8)$$

Promise $a_i \overset{-d}{\rightarrow} a_j$

Let $C_{ij}^{(-d)}$ be the rate at which a_i can receive data from a_j. Then the time to receive the data is:

$$t\left(a_i \overset{-d}{\rightarrow} a_j\right) = \frac{Q_j}{C_{ji}^{(-d)}}. \tag{9}$$

Agent constraints on $\pm d$

It must be generally true that the rate at which a promise is used is less than or equal to the rate at which it is provided.

$$C_{ji}^{(-b)} \leq C_{ij}^{(+b)} \tag{10}$$

for any promise body b. Moreover, if we let $C_i^{(\pm b)}$ be the maximum communication capacities of the agents for sending/receiving (a property of the agents rather than the channel between them) we must have the additional constraint that, if an agent has several neighbours, the sum of communications with all neighbours cannot exceed the agent's own capacity:

$$\sum_j A_{ij}^{(-b)} C_{ij}^{(-b)} \leq C_i^{(-b)}, \quad \sum_j (A_{ij}^{(+b)})^{\mathrm{T}} C_{ij}^{(+b)} \leq C_i^{(+b)} \tag{11}$$

Promise $a_i \overset{+a}{\rightarrow} a_j$ (aggregation)

Each node that receives data from more than one source promises to aggregate the information according to some algorithm, which we shall not specify in any more detail. We define the aggregation to be the computation

$$a_i \equiv q_i + \sum_k A_{ij}^{(-d)} a_j, \quad i, j = 1 \ldots N. \tag{12}$$

For leaf nodes, the second term is zero and the aggregation is simply equal to the contribution from the agent node itself. The recipient of this promise might be the aggregation agent above the promiser in the aggregation tree, or it could be an agent external to the tree, such as a policy agent or an external observer. The final recipient at the top of the tree makes this promise either to itself or to a suitable policy agent or observer.

This aggregation promise has a number of consequences. The result of the computation is dependent on the values collected by the $-d$ promises of neighbours. Thus there must be a strict ordering of events before the promise can be completed in a given time frame. We must always remember that the aggregating agent knows only what it has been promised in a scheme of voluntary cooperation.

The semantics of the aggregation promise must now be defined. Several alternatives present themselves:

1. The node polls its sources in turn.
2. The node aggregates from its sources in parallel.
3. The node does not promise its result until all sources have kept their promises to provide data, i.e. we have conditional promises:

$$a_{i-1} \overset{+d/a}{\rightarrow} a_i,\ a_i \overset{a/-d}{\rightarrow} a_{i+1} \tag{13}$$

4. The node does not wait for its sources and provides its best answer and there is no condition on the dependents in the chain:

$$a_{i-1} \overset{+d}{\rightarrow} a_i,\ a_i \overset{a}{\rightarrow} a_{i+1} \tag{14}$$

These details are important to fully describe the propagation of data within a network structure.

The time to complete the aggregation promise is straightforwardly given by:

$$t\left(a_i \overset{+a}{\rightarrow} a_j\right) = \frac{Q_i + \sum_k A_{ik}^{(-d)} Q_k}{C_i^{(a)}}. \tag{15}$$

5.4 Scaling Behaviour

The two extreme cases of network pattern are the star network (maximum width) and the chain (maximum depth), see fig. 26. The chain also mathematically half a ring, thus we can also model token rings. What is the scalability of these structures? We cannot any longer characterize the scaling as a function of the total number of nodes, except in the extreme cases of chain and star. We can say broadly that increased pattern width (average node degree) tends to allow greater *flow rate*, but each constriction limits the flow and increases the cost to the aggregator. Increased depth increases the *flow time*.

From the constraint expressions, we see that only local behaviour is important. The key scaling relationships are the same as those identified for the extreme star topology [41, 42], but these apply not for the total number of nodes N, but for the local node degree k. However, the time uncertainty in the data was not discussed in those papers.

Consider a sampling/aggregation process scheduled to run every P seconds along a chain, where P is the period (see fig. 27), so that the promise compliance frequency is P. In a periodic process, that which happens in one period is identical (for all intents and purposes) to that which happens in any period. Thus, if we describe one period, we have described all of them. We can thus turn to a description of the system in terms of a new time variable τ which runs from time 0 to P. These are related by

$$\begin{aligned} t &= nP + \tau,\ n = 0, \pm 1, \pm 2, \ldots & (16) \\ \tau &= t \bmod P. & (17) \end{aligned}$$

The effects of scheduling become clearest when we use modulo "clock" arithmetic τ. Let t_n be the real time at which agent a_n wakes up, samples its data and collects data from

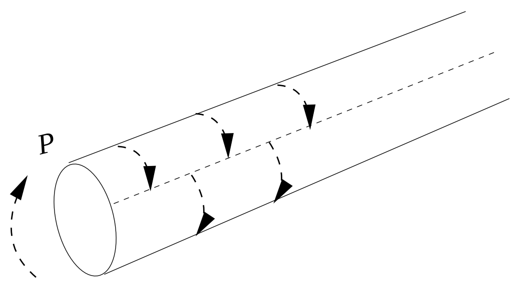


Figure 27: Agents operating in periodic sampling mode.

upstream agent a_{n-1}. The time to keep the promise $t(a_n \overset{-d}{\rightarrow} a_{n-1}) = t_n - t_{n_1}$. In order for promises to be kept data to be transmitted along the chain in a single avalanche, we must have:

$$t_n > t_{n-1},\ \forall n > 1. \tag{18}$$

i.e. each successive agent must begin at a time that is strictly greater than that of its predecessor. This constraint is easily solved by choosing:

$$t_n = t_0 + (n-1)t_s, \tag{19}$$

for some offset t_s. The subtlety here is that this schedule has to be wound around the periodic time schedule of the agents. Since each agent is time-shifted by t_s relative to the last, the total time-span of a single schedule is $(N-1)t_s$. As long as this is less than a single period, there will be no phasing between the delays and the periodic activation of the agents. However, in long chains where $(N-1)t_s > P$ some agents will wrap around into the next activation of the schedule and possible start out of sequence with respect to periodic time τ. Thus, even if eqns. (18) and (19) are satisfied in real time, it is not necessarily true that $\tau_n > \tau_{n-1}$. Indeed, this condition will necessarily be violated for some of the agents as long as $(N-1)t_s > P$.

5.5 Experimental Results

We evaluated the performances of the star pattern (maximum width network pattern) through an extensive set of experiments in the context of a load-balancing architecture [49]. In particular, we were interested in comparing this network pattern with two different strategies: obligation strategy (push-based) and voluntary strategy (pull-based). Traditional load balancing scheme are a typical instantiation of the obligation push-based strategy, where the decision of whether a server should receive a request or not belongs exclusively to the load balancer. Autonomics sceptics often imagine that this kind of approach is fundamental to the idea of "control" and the idea of component autonomy stands in the way of proper resource sharing if servers will not do as they are told by an authoritative controller. We have already argued against this viewpoint [47] and show this belief to be erroneous below.

At the opposite, the star pattern using voluntary strategy (see fig. 26) can be instantiated as a pull-based load balancing architecture, where autonomous servers can manage load at their own convenience. Each server knows its own capabilities and state more quickly and accurately than any external monitor, so it seems reasonable to explore the idea that it is the best judge of its own performance. As service requests are pulled from a load balancer, servers can decide to take on work depending not only on their available resources but also on its own internal parameters including their willingness to interact. The authoritative decision of the load balancer is therefore transferred to the autonomous servers.

We compared these two strategies using the open source suite for queuing network modeling and workload analysis [50] developed by the Performance Evaluation Lab at the Polytechnic University of Milan. This simulation tool can typically be used to experiment traditional push-based schemes. We extended it so that we can model the behavior of a pull-based load balancer with direct server return. We considered during the experiments a system composed of a load balancer L at the front office and a set of n servers $S = \{S_1, S_2, ..., S_n\}$ at the back office. The load balancer L implements either a push-based scheme (obligation) or a pull-based scheme (voluntary) depending on the scenario. We assume that the arrival and completion process distributions follow a typical Poisson distribution in discrete time. Despite the controversy regarding the inter-arrival times, we use Poisson distributed arrivals, this allows a direct comparison with ref. [51] and we would not expect the choice to affect the broad conclusions of our results.

We modeled a system with three servers $\{S_1, S_2, S_3\}$ of same capabilities. Each server is capable of processing an average of 100 requests per second. We measured the aver-

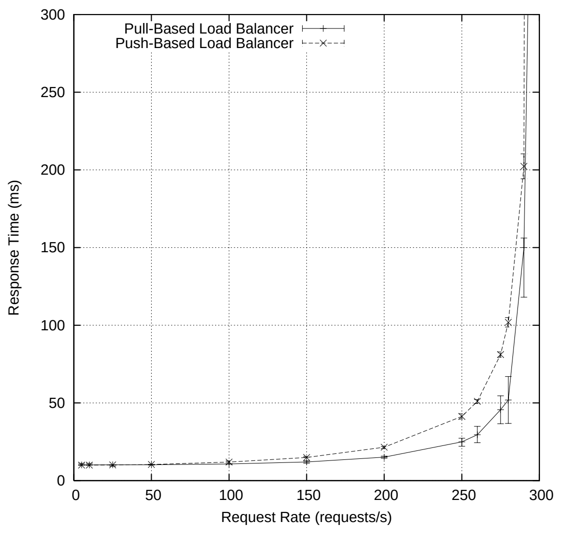
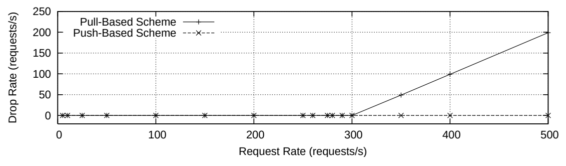
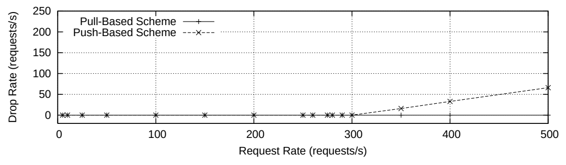


Figure 28: Star Pattern - Response Time in the Context of a Load Balancing Architecture with Homogeneous Servers, by using Obligation (plain line) and Voluntary (dotted line) Regimes

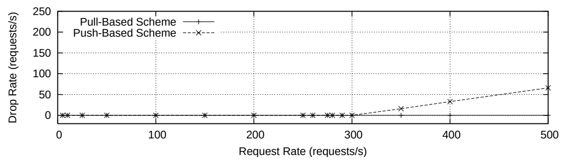
age response time obtained with the pull-based load balancer while varying the request rate from 5 to 300 requests per second. We assume here the natural notion of response time perceived by users, that is, the time interval between the instant of the submission of a request and the instant the corresponding reply arrives completely at the user. We compared these values with the response time provided by a push-based load balancer implementing the classic round-robin scheme where servers are taken each in turn with-



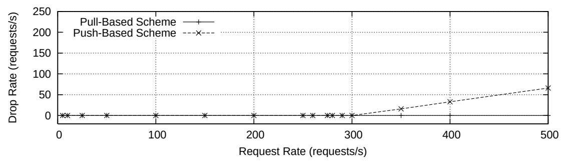
(a) Load Balancer



(b) Server S_1



(c) Server S_2



(d) Server S_3

Figure 29: Star Pattern - Drop Rates in the Context of a Load Balancing Architecture with Homogeneous Servers, by using Obligation (plain line) and Voluntary (dotted line) Regimes

out consideration of their current queue length or latency. The experimental results are summarized in figure 28. The horizontal axis corresponds to the different request rate values. We plotted for each of them the average response time obtained with the pull-based load balancer (plain line) compared to the one obtained with the push-based load balancer (dotted line).

The two graphs show the same tendency when we increase the request rate: the response time first stays relative low and stable (on the left of the figure), then grows exponentially when we tends to 300 requests per second (on the right of the figure). This value corresponds to the theoretical maximal processing rate of the system (the sum of three servers capable of processing 100 requests per second each). When comparing the two star pattern strategies, we can observe that the performance is quite similar at a low response rate (from 5 to 100 requests per second). The push-based load balancer is even better than the pull-based strategy at the lowest rates. However, at high response rates (from 100 requests per second), we clearly see that the pull-based load balancer produces faster response times than the push-based load balancer.

Additional parameters need to be taken into account to complete and refine this comparison. In particular, we measured the request drop rate while varying the request rate values. The drop rate is the number requests per second never processed at all by an element of the system. We plotted these rates in figure 29 for respectively the load balancer and each of the three servers. When looking at these results, we first can observe that the system starts to drop requests at around 300 requests per second (compare to [51]), when the request rate exceeds the maximal processing rate. However, the dropped requests are not distributed in the same manner in the system. In the obligation scenario (dotted line), the dropped requests are equally distributed among the three servers. The load balancer does not drop any requests during the experiments, even with high request rates. The reason for these results is that all the requests are dispatched by the load balancer to one of the servers whatever the server is overloaded or not (i.e. it simply pushes the problem downstream).

Intuitively, the pull based mechanisms needs a longer queue at the dispatcher since this is where waiting builds up at the system bottleneck, but this does not imply that greater resources are needed. The pull strategy is like an airport check-in queue: servers pull passengers from a single line that fills the same space as would multiple lines. The same resources are simply organized differently. Moreover, the reliance on a single dispatcher need not be a problem is redundancy is factored into the calculation [52].

As the servers have homogeneous capabilities and the load balancer takes servers each in turn, the dropped rate is almost identical for the three servers and corresponds to the third of the total drop rate in the system. In the pull-based scenario (plain line), requests are only dropped by the load balancer. The servers process requests on demand by pulling the load balancer. As a consequence, the servers are not overloaded and do not need to drop requests during the experiments. The requests are waiting at the load balancer and are directly dropped by it when the request rate is too high for the system.

5.6 Conclusions

We address in this section the scalability issue of peer services employing voluntary cooperation, in the context of Network Patterns (tree-like overlay networks). First, we have modeled these patterns using the framework of promise theory. Instead of considering an obligation regime, where requests are pushed from one or more nodes through the network, we have preferred a distributed pull-based scheme, where nodes can cooperate with others on a voluntary basis. We have then analyzed the resulting promise graph in order to determine the kinematical properties and the scaling behaviour of Network Patterns. In particular, we have shown that (1) increasing pattern width tends to allow greater flow rate, but each constriction limits the flow and increases the cost to the aggregator, and (2) increasing depth increases the flow time. We have also evaluated the performance of the star pattern through an extensive set of experiments in the context of a load balancing architecture. We have observed the pull-based strategy provides better results than the traditional push-based strategy (obligation) with autonomous servers. Our future work will focus on the analysis of the promise propagation time for chain patterns and on complementary experiments with the star pattern for determining the impact of heterogeneous servers and for quantifying the bottleneck effect.

6 Conclusions

We report in this deliverable on the activities in the work-package 7 performed in the time frame July 2007 to March 2008. These activities have been driven by three main tasks, that emerged from an open call for proposals. The first topic, addressed in the activity NMTA (Network Management Trace Analysis), is centered on precisely defining the key concepts and terminology for SNMP trace analysis. The second topic is addressing the scalable and distributed security monitoring. In the SNID task (Scalability of Network Intrusion Detection), we addressed the issues on how flow based monitoring is dependent of the network scale and sampling approach and covered the algorithmic graph based methods required to monitor large network telescopes and honeypots. The third topic is concerned with a new configuration paradigm based on voluntary peer to peer contribution.This work proposes a radically new theoretical concept of using the notion of promise in developing large scale configuration operations. The scientific impact of these activities is high. IETF drafts resulted and several paper submissions to conferences are the results of them.

7 Acknowledgement

This deliverable was made possible due to the large and open help of the WP7 Partners of the EMANICS NoE. Many thanks to all of them.

References

[1] J. Case, R. Mundy, D. Partain, and B. Stewart. Introduction and Applicability Statements for Internet Standard Management Framework. RFC 3410, SNMP Research, Network Associates Laboratories, Ericsson, December 2002.

[2] D. Harrington, R. Presuhn, and B. Wijnen. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411, Enterasys Networks, BMC Software, Lucent Technologies, December 2002.

[3] J. Schönwälder. SNMP Traffic Measurements and Trace Exchange Formats. Internet Draft (work in progress) <draft-irtf-nmrg-snmp-measure-04.txt>, Jacobs University Bremen, March 2008.

[4] R. Presuhn. Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP). RFC 3416, BMC Software, December 2002.

[5] J. Schönwälder, A. Pras, M. Harvan, J. Schippers, and R. van de Meent. SNMP Traffic Analysis: Approaches, Tools, and First Results. In *Proc. 10th IFIP/IEEE International Symposium on Integrated Network Management*, May 2007.

[6] G. van den Broek, J. Schönwälder, A. Pras, and M. Harvan. SNMP Trace Analysis Definitions. Internet Draft (work in progress) <draft-schoenw-nmrg-snmp-trace-definitions-00.txt>, University of Twente, Jacobs University Bremen, ETH Zurich, January 2008.

[7] Cisco IOS NetFlow Configuration Guide, Release 12.4. http://www.cisco.com, February 2008.

[8] M. Fomenkov, K. Keys, D. Moore, and K. Claffy. Longitudinal study of internet traffic in 1998-2003. In *WISICT '04: Proceedings of the winter international synposium on Information and communication technologies*, pages 1–6. Trinity College Dublin, 2004.

[9] E.J.Vlieg. Representativiness of internet2 usage statistic. In *8th Twente Student Conference on Information Technology*, pages 25–29, January 2008.

[10] SURFnet. www.surfnet.nl, February 2008.

[11] Géant. www.geant.net, February 2008.

[12] Cisco IOS NetFlow. http://www.cisco.com/go/netflow, February 2008.

[13] M. Siekkinen, E.W. Biersack, G. Urvoy-Keller, V. Goebel, and T. Plagemann. Intrabase: integrated traffic analysis based on a database management system. *End-to-End Monitoring Techniques and Services, 2005. Workshop on*, pages 32–46, 15 May 2005.

[14] T. Fioreze, M. Oude Wolbers, R. van de Meent, and A. Pras. Finding elephant flows for optical networks. In *Integrated Network Management*, pages 627–640. IEEE, 2007.

[15] Robin Sommer and Anja Feldmann. Netflow: information loss or win? In *Internet Measurement Workshop*, pages 173–174. ACM, 2002.

[16] Gerhard Munz and Georg Carle. Real-time analysis of flow data for network attack detection. In *Integrated Network Management*, pages 100–108. IEEE, 2007.

[17] IPTV@UT: Digital television at the campus. http://iptv.utwente.nl/, February 2008.

[18] Carey E. Priebe, John M. Conroy, David J. Marchette, and Youngser Park. Scan statistics on enron graphs. *Comput. Math. Organ. Theory*, 11(3):229–247, 2005.

[19] Mark Burgess. *Analytical network and system administration*. John Wiley & Sons Ltd., 2004.

[20] Fabien Pouget, Marc Dacier, and Hervé Debar. Attack processes found on the Internet. In *NATO Research and technology symposium IST-041/RSY-013 "Adaptive Defence in Unclassified Networks", 19 April 2004, Toulouse, France*, Apr 2004.

[21] V. Yegneswaran, P. Barford, and D. Plonka. The design and use of internet sinks for network abuse monitoring, 2004.

[22] Colleen Shannon, David Moore, and Emile Aben. The caida backscatter-2004-2005 dataset - may 2004 - november 2005, http://www.caida.org/data/passive/backscatter_2004_2005 _dataset.xml.

[23] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet Denial of Service : Attack and Defense Mechanisms*. Radia Perlman Computer Networking and Security. Prentice Hall PTR, december 2004.

[24] B. Plattner D. Brauckhoff, M. May. Flow-level anomaly detection - blessing or curse?, May 2007. IEEE INFOCOM 2007, Student Workshop, Anchorage, Alaska, U.S.A.

[25] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, 2000.

[26] Defcon capture the flag traffic logs. `http://cctf.shmoo.com/`.

[27] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[28] CSG Testbed Infrastructure for Research Activities. `http://www.csg.uzh.ch/services/testbed/`.

[29] Robin Sommer. Bro: An open source network intrusion detection system. `http://citeseer.ist.psu.edu/743026.html`.

[30] M. Roesch. Snort-Lightweight Intrusion Detection for Networks. *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, 1999.

[31] Stefan Kornexl. High-performance packet recording for network intrusion detection. Diplomarbeit, Technische Universitt Mnchen, Munich, Germany, January 2005.

[32] Broccoli: The bro client communications library. `http://www.icir.org/christian/broccoli/manual/index.html`.

[33] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale Botnet Detection and Characterization. *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots 07)*, 2007.

[34] Schneier on security: The storm worm. `http://www.schneier.com/blog/archives/2007/10/the_storm_worm.html`.

[35] P. Porras, H. Saıdi, and V. Yegneswaran. A Multi-perspective Analysis of the Storm (Peacomm) Worm.

[36] Peacomm.c, cracking the nutshell. `http://www.antirootkit.com/articles/eye-of-the-storm-worm/Peacomm-C-Cracking-the-nutshell.html`.

[37] Bleeding edge threats. `http://www.bleedingthreats.net/`.

[38] D. Dagon, G. Gu, C. Zou, J. Grizzard, S. Dwivedi, W. Lee, and R. Lipton. A Taxonomy of Botnets. *Proceedings of CAIDA DNS-OARC Workshop, San Jose, CA (July 2005)*.

[39] F. Constantinou and P. Mavrommatis. Identifying Known and Unknown Peer-to-Peer Traffic. *Proc. of Fifth IEEE International Symposium on Network Computing and Applications*, pages 93–102, 2006.

[40] Securitypronews: Storm botnets using encrypted traffic. `http://www.securitypronews.com/insiderreports/insider/spn-49-20071016StormBotnetsUsingEncryptedTraffic.html`.

[41] M. Burgess and G. Canright. Scalability of peer configuration management in partially reliable and ad hoc networks. *Proceedings of the VIII IFIP/IEEE IM conference on network management*, page 293, 2003.

[42] M. Burgess and G. Canright. Scaling behaviour of peer configuration in logically ad hoc networks. *IEEE eTransactions on Network and Service Management*, 1:1, 2004.

[43] K-S Lima and R. Stadler. A navigation pattern for scalable internet management. *Proceedings of the VII IFIP/IEEE IM conference on network management*, 2001.

[44] A. Gonzalez Prieto and R. Stadler. Adaptive distributed monitoring with accuracy objectives. *ACM SIGCOMM workshop on Internet Network Management (INM 06), Pisa, Italy*, 2006.

[45] M. Dam and R. Stadler. A generic protocol for network state aggregation. *RVK 05, Linkping, Sweden, June 14-16*, 2005.

[46] Richard Murch. *Autonomic Computing.* IBM Press, 2004.

[47] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97–108, 2005.

[48] M. Burgess. *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester, 2004.

[49] R. Badonnel and M. Burgess. Dynamic Pull-Based Load Balancing for Autonomic Networks. San Jose, USA, April 2008. Short Paper, Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2008).

[50] M. Bertoli, G. Casale, and G. Serazzi. An Overview of the JMT Queueing Network Simulator. Technical Report TR 2007.2, Politecnico di Milano - DEI, 2007.

[51] M. Burgess and G. Undheim. Predictable scaling behaviour in the data centre with multiple application servers. In *Lecture Notes on Computer Science, Proc. 17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006)*, volume 4269, pages 9–60. Springer, 2006.

[52] D. A. Menasc, V. A .F Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Prentice Hall, Upper Saddle River, New Jersey 07458, first edition edition, 1994.